Calculabilité

Pascal MANOURY

11 novembre 2006

Donner une idée de la notion de «calculable» ; montrer d'indécidabilité du problème de l'arrêt.

«Calculabilité» (en anglais *«computability»*) est le terme choisi par A. Turing pour recouvrir les notions de *«mécanicité»* et d'*«effectivité»* des processus de calcul et de preuve en mathématique qu'il étudia et précisa dans son article de 1936 (*«On Computable Numbers With an Application to the Entscheidungsproblem»*) en définissant ses fameuses *«Logical Computing Machines»* connues aujourd'hui sous le nom de *«machines de Turing»*. Pour A. Turing, est *«calculable»* ce qui l'est par ses machines. De son côté, A. Church proposait un autre modèle de calcul basé sur un formalisme de notation des fonctions mathématiques : le *«λ-calcul»* (lire *«lambda calcul»*). Tous deux montrèrent que l'ensemble des fonctions calculables selon leurs modèles coïncide avec l'ensemble des *«fonctions récursives générales»* élaboré par J. Herbrand/K. Gödel/W. Ackermann.

L'équivalence des modèles renforce le bien fondé du concept de *«calculabilité»* et laisse à penser que celui-ci a été correctement cerné : c'est la *«thèse de Church-Turing»*. On dispose donc d'un moyen d'effectuer tous les calculs possibles, mais cela ne signifie par pour autant que tout est calculable : au contraire, il y aura toujours un calcul que la machine ne saura faire.

Nous introduisons dans cette note la notion de calculabilité en en donnant deux modèles : les fonctions récursives générales et les *«machines à registres illimités»* qui sont un modèle de calcul plus récent et plus proche des *«machines à calculer»* que sont les ordinateurs contemporains que ne le sont les machines de Turing. Nous montrons l'équivalence des deux modèles proposés. Nous prouvons brièvement en fin de note que la solution du problème de savoir si un programme (pour une machine à registre) est susceptible de boucler ou non n'est pas calculable : c'est *«l'indécidabilité du problème de l'arrêt»*.

1 Deux modèles de calculabilité

1.1 Fonctions récursives

L'ensemble des fonctions *«récursives»* est le plus petit ensemble de fonctions qui contient

R1 les *«fonctions constantes»* : $c(x_1, \ldots, x_k) = 0$

R2 la fonction *«successeur»* : $s(x) = x + 1$

R3 les *«projections»* : $p_i(x_1, \ldots, x_k) = x_i$ avec $1 \leq i \leq k$

et qui est clos par

R4 *«composition»* : soient $h, g_1, \ldots, g_n$ des fonctions récursives, la fonction f telle que

$$f(x_1, \ldots, x_k) = h(g_1(x_1, \ldots, x_k), \ldots, g_n(x_1, \ldots, x_k))$$

est récursive.

R5 *«schéma primitif récursif»* : soient h et g deux fonctions récursives, la fonction f telle que

$$\begin{array}{lcl} f(0, x_1, \ldots, x_k) & = & h(x_1, \ldots, x_k) \\ f(x+1, x_1, \ldots, x_k) & = & g(x, x_1, \ldots, x_k, f(x, x_1, \ldots, x_k)) \end{array}$$

est récursive.

R6 et *«schéma de minimisation»* : soit g une fonction, on note $\mu x.(g(x_1,\ldots,x_k,x)=0)$ le *plus petit* x tel que $g(x_1, dots, x_k, x) = 0$. Notons qu'un tel x peut ne pas exister.
Soit g une fonction récursive, la fonction f définie par

$$f(x_1,\ldots,x_k) = \mu x.(g(x_1,\ldots,x_k,x)=0)$$

est récursive.

1.1.1 Primitives récursives

Les fonctions récursives définies en suivant uniquement les clauses R1 à R5 sont dites *primitives récursives*. Plus généralement, une fonction f est dite primitive récursive dés lors qu'il existe une fonction f' définie selon les clauses R1-R5 telle que pour tout x, $f'(x) = f(x)$.

L'addition est primitive récursive. En effet l'addition est définie par les équations

$$\begin{array}{lcl} ad(0,y) & = & y \\ ad(x+1,y) & = & s(ad(x,y)) \end{array}$$

qui ressemblent fort au schéma primitif récursif. Pour le retrouver rigoureusement, on pose $g(y) = p_1(y) = y$ (qui est primitive récursive) et $h(x,y,z) = s(p_3(x,y,z)) = s(z)$ (qui est aussi primitive récursive). Les équations de ad deviennent alors

$$\begin{array}{lcl} ad(0,y) & = & g(y) \\ ad(x+1,y) & = & h(x,y,ad(x,y)) \end{array}$$

qui répondent exactement au schéma primitif récursif.

La multiplication est primitive récursive. La multiplication est définie par

$$\begin{array}{lcl} mul(0,y) & = & 0 \\ mul(x+1,y) & = & ad(mul(x,y),y) \end{array}$$

Pour s'en convaincre, on prend $g(y) = c(y) = 0$ et $h(x,y,z) = ad(p_3(x,y,z), p_2(x,y,z)) = ad(z,y)$.

Remarque:
- on s'autorisera à utiliser les notations infixes usuelles des fonctions d'addition et de multiplication : $x+y$ et $x \cdot y$. Il en ira de même pour les autres fonctions arithmétiques usuels : soustraction, tests d'égalité, infériorité, etc. ;
- on ne mentionnera plus les usages triviaux des fonctions constantes, des projection ou de la composition dans les schémas de définition.

Prédécesseur inverse de la fonction successeur, sauf sur 0 dont, par convention, le prédécesseur est 0. La fonction *«prédécesseur»*, définie par

$$\begin{array}{lcl} pred(0) & = & 0 \\ pred(x+1) & = & x \end{array}$$

est primitive récursive.

Fonctions booléennes et alternative Les valeurs booléennes *«vrai»* et *«faux»* sont représentées (on dit aussi *«codées»*), respectivement, par n'importe quel entier non nul et 0.

On définit les fonctions (primitives récursives) *not*, *and* et *or* représentant les fonctions de négation, conjonction et disjonction

$$\begin{array}{lcl} not(0) & = & 1 \\ not(x+1) & = & 0 \end{array}$$

$$and(x_1, x_2) = x_1 \cdot x_2$$

$$or(x_1, x_2) = x_1 + x_2$$

On pourra noter $x_1 \wedge x_2$ pour $and(x_1, x_2)$ et $x_1 \vee x_2$ pour $or(x_1, x_2)$.

La fonction (primitive récursive) alternative (ou *«conditionnelle»*) est définie par les équations

$$\begin{array}{lcl} if(0, x_1, x_2) & = & x_2 \\ if(x+1, x_1, x_2) & = & x_1 \end{array}$$

L'alternative permet des définitions *«par cas»* :

$$f(x_1, \ldots, x_k) = if(t(x_1, \ldots, x_k), f_1(x_1, \ldots, x_k), f_2(x_1, \ldots, x_k))$$

Si t, f_1 et f_2 sont primitives récursives alors f également. Pour reprendre l'usage, on notera l'équation ci-dessus comme ci-dessous :

$$\begin{array}{lcll} f(x_1, \ldots, x_k) & = & f_1(x_1, \ldots, x_k) & \text{si } t(x_1, \ldots, x_k) \\ & = & f_2(x_1, \ldots, x_k) & \text{sinon} \end{array}$$

Si la fonction f_2 est à son tour une alternative, on utilisera le même racourci d'écriture

$$\begin{array}{lcll} f(x_1, \ldots, x_k) & = & f_1(x_1, \ldots, x_k) & \text{si } t_1(x_1, \ldots, x_k) \\ & = & f_2(x_1, \ldots, x_k) & \text{si } t_2(x_1, \ldots, x_k) \\ & = & f_3(x_1, \ldots, x_k) & \text{sinon} \end{array}$$

ad libitum pour toute imbrication d'alternatives de la forme $if(t_1, x_1, if(t_2, x_2, if(t_3, x_3, \ldots)\ldots)$.

Itération l'*«itérée»* f^n d'une fonction f primitive récursive est primitive récursive. En effet, l'itérée n-ième de f est définie par induction sur n : $f^0(x) = x$ et $f^{n+1}(x) = f(f^n(x))$. Soit alors f^* telle que

$$\begin{array}{lcl} f^*(0, y) & = & y \\ f^*(x+1, y) & = & f(f^*(x, y)) \end{array}$$

Si f est primitive récursive, alors f^* également, et on vérifie aisément par induction sur n que $f^n(x) = f^*(n, x)$.

Voici une seconde forme d'itération où f est une fonction binaire :

$$\begin{array}{lcl} f^{**}(0, y) & = & y \\ f^{**}(x+1, y) & = & f(x, f^{**}(x, y)) \end{array}$$

Le *«développé»* d'une telle itération est $f(x, f(x-1, \ldots f(1, f(0, y))\ldots))$. f^{**} est primitive récursive si f l'est.

Cette seconde forme d'itération permet de voir que la somme et le produit bornés sont primitifs récursifs :

$$\begin{array}{lcl} \Sigma_{i=0}^{n} f(i) & = & S^{**}(n, 0) \\ \Pi_{i=0}^{n} f(i) & = & P^{**}(n, 1) \end{array}$$

avec $S(x, y) = f(x) + y$ et $P(x, y) = f(x) \cdot y$.

La somme et le produit bornés permettent de définir des fonctions primitives récursives de test pour les quantifications bornées :

$$\begin{array}{lcl} \exists z \leq x.(f(x_1,\ldots,x_k)=0) & = & \Sigma_{i=0}^{x}.(f(x_1,\ldots,x_k)=0) \\ \forall z \leq x.(f(x_1,\ldots,x_k)=0) & = & \Pi_{i=0}^{x}.(f(x_1,\ldots,x_k)=0) \end{array}$$

En effet, pour l'existensielle, il suffit qu'un des $f(x_1,\ldots,x_k)=0$ soit non nul pour que la somme soit non nulle ; pour l'universelle, il faut que tous les $f(x_1,\ldots,x_k)=0$ soient non nuls pour que le produit soit non nul.

Modification de paramètres soient g, h et i des fonctions primitives récursives. La fonction f définie par

$$\begin{array}{lcl} f(0,y) & = & g(y) \\ f(x+1,y) & = & h(x,y,f(x,i(y))) \end{array}$$

est primitive récursive.

Ce schéma n'est pas primitif à cause du $i(y)$ dans l'appel récursif. Cependant, si l'on *«déroule»* le calcul de f en suivant les équations, on obtient

$$\begin{array}{rcl} f(x+1,y) & = & h(x,y,f(x,i(y)) \\ & = & h(x,y,h(x-1,i(y),f(x-1,i^2(y)))) \\ & = & h(x,y,h(x-1,i(y),h(x-2,i^2(y),f(x-2,i^3(y))))) \\ & \vdots & \\ & = & h(x,y,h(x-1,i(y),h(x-2,i^2(y),\ldots f(1,i^{x-1}(y)))\ldots) \\ & = & h(x,y,h(x-1,i(y),h(x-2,i^2(y),\ldots,h(1,i^{x-1}(y),f(0,i^x(y)))\ldots))) \\ & = & h(x,y,h(x-1,i(y),h(x-2,i^2(y),\ldots,h(1,i^{x-1}(y),g(i^x(y)))\ldots))) \end{array}$$

On remarque dans ce développement que pour chaque application de h, si le premier argument est $x-k$, le deuxième est $i^k(y)$. En d'autres termes, si x_0 est la valeur initiale de x, à chaque application de h, on a $h(x,i^{x_0-x}(i),\ldots)$. Posons $h'(x,x_0,y,z)=h(x,i^{x_0-(x+1)}(y),z)$. On note également que l'argument passé à g est $i^{x_0}(y)$. Soit alors f' telle que

$$\begin{array}{lcl} f'(0,x_0,y) & = & g(i^{x_0}(y)) \\ f'(x+1,x_0,y) & = & h'(x,x_0,y,f'(x,x_0,y)) \end{array}$$

qui est primitive récursive.

On peut montrer par récurence sur x que pour tout k, $f'(x,x+k,y)=f(x,i^k(y))$ (exercice). On pose alors $f(x,y)=f'(x,x,y)$ ce qui fait de f une fonction primitive récursive.

En faisant appel à la seconde forme d'itération donnée ci-dessus, on peut vérifier que la forme plus générale de définition avec modification de paramètre

$$\begin{array}{lcl} f(0,y) & = & g(y) \\ f(x+1,y) & = & h(x,y,f(x,i(x,y))) \end{array}$$

est primitive récursive (exercice).

Exercice: Soient g,h_1,i_1,h_2,i_2,r primitives récursive, montrer que la fonction f telle que

$$\begin{array}{lcll} f(0,y) & = & g(y) & \\ f(x+1,y) & = & h_1(x,y,f(x,i_1(x,y))) & \text{si } r(x,y)=0 \\ & = & h_2(x,y,f(x,i_2(x,y))) & \text{sinon} \end{array}$$

est primitive récursive.

Soustraction, comparaisons La *«soustraction entière»* telle que $x - y = 0$ si $x \leq y$ est définie par

$$\begin{array}{lcl} 0 - y & = & 0 \\ (x+1) - 0 & = & x+1 \\ (x+1) - (y+1) & = & x - y \end{array}$$

Cette définition utilise la *«modification de paramètre»* ($y+1$ devient y dans la dernière équation) ainsi qu'un développement *«par cas»* de ce paramètre. On peut ramener à un schéma connu ce développement par cas :

$$\begin{array}{lcll} 0 - y & = & 0 & \\ (x+1) - y & = & x+1 & \text{si } y = 0 \\ & = & x - pred(y) & \text{sinon} \end{array}$$

On se ramène ainsi au simple changement de paramètre et la fonction définie est primitive récursive. Nous nous autoriserons souvent un tel racourci de notation mêlant test à 0 et modification évidente de paramètre.

Sur un schéma analogue, on définit les relations d'égalité, d'infériorité et de supériorité comme fonctions entre entiers :

$$\begin{array}{lcl} eq(0,0) & = & 1 \\ eq(0,y+1) & = & 0 \\ eq(x+1,0) & = & 0 \\ eq(x+1,y+1) & = & eq(x,y) \end{array}$$

$$\begin{array}{lcl} le(0,y) & = & 1 \\ le(x+1,0) & = & 0 \\ le(x+1,y+1) & = & le(x,y) \end{array}$$

$$\begin{array}{lcl} ge(0,y) & = & 0 \\ ge(x+1,0) & = & 1 \\ ge(x+1,y+1) & = & le(x,y) \end{array}$$

Ces fonctions sont primitives récursives. On notera simplement $x = y$, $x \leq y$ et $x \geq y$ pour $eq(x,y)$, $le(x,y)$ et $ge(x,y)$.

On déduit des tests d'inégalité les fonctions (primitives récursives) min et max qui donnent, respectivement, le minimum et le maximum de deux entiers :

$$\begin{array}{lcll} min(x,y) & = & x & \text{si } x \leq y \\ min(x,y) & = & y & \text{sinon} \end{array}$$

$$\begin{array}{lcll} max(x,y) & = & x & \text{si } x \geq y \\ max(x,y) & = & y & \text{sinon} \end{array}$$

On vérifie que les résultats sont corrects lorsque $x = y$.

Récurence plus générale soit une fonction primitive récursive p telle que $p(x) \leq x$. Soit alors f définie par

$$\begin{array}{lcl} f(0,y) & = & g(y) \\ f(x+1,y) & = & h(x,y,f(p(x),y)) \end{array}$$

Nous allons voir que f est primitive récursive. Soit f' définie par

$$\begin{array}{lcll} f'(0,x',x_0,y) & = & g(y) & \\ f'(x+1,x',x_0,y) & = & h(x_0,y,f'(x,p(x),x,y)) & \text{si } x+1 = x' \\ & = & f'(x,x',x_0,y) & \text{sinon} \end{array}$$

qui est primitive récursive, et on a $f(x,y) = f'(x,p(x),x,y)$ (exercice).

Minimisation bornée c'est le schéma de minimisation (R6) où l'on ajoute la contrainte que la valeur cherchée doit être inférieure à une valeur donnée : $\mu z \leq x.(f(x_1,\ldots,x_k,z)=0)$. Si un tel z n'existe pas, on pose par convention que la valeur obtenue est 0. Soit f'

$$\begin{array}{lcll} f'(x_1,\ldots,x_k,0) & = & 0 & \\ f'(x_1,\ldots,x_k,z+1) & = & f'(x_1,\ldots,x_k,z) & \text{si } \Sigma_{i=0}^{z}(f(x_1,\ldots,x_k,i)=0)\geq 1 \\ & = & z+1 & \text{si } f(x_1,\ldots,x_k,z+1)=0 \\ & = & 0 & \text{sinon} \end{array}$$

est primitive récursive et on pose $\mu z \leq x.(f(x_1,\ldots,x_k,z)=0) = f'(x_1,\ldots,x_k,x)$.

Remarque: il faut comprendre le premier cas de la seconde équation ainsi : si le produit $\Sigma_{i=0}^{z}(f(x_1,\ldots,x_k,i)=0)$ n'est pas nul, c'est qu'il existe au moins un $i<z+1$ tel que $f(x_1,\ldots,x_k,i)=0$; même si $f(x_1,\ldots,x_k,z+1)=0$, ce $z+1$ n'est pas le plus petit et il faut continuer à chercher. L'ordre des cas est significatif : le deuxième n'est envisagé que si le premier échoue (*cf* alternative) ; dès lors, le $z+1$ trouvé au deuxième cas est celui cherché (le plus petit tel que ...).

1.2 Machines à registres

Une *«machine à registres»* est un modèle idéal de processeur qui exécute un programme en mettant en œuvre un nombre illimité de registres. Chaque registre peut contenir un entier aussi grand que l'on veut. Un *«état»* d'une machine à registres est défini par

- une suite finie d'entiers $r_0, r_1, \ldots, r_m$ représentant les *«registres»* (i.e ; la mémoire) de la machine (il y a toujours au moins l registre : r_0) ;
- une suite finie non vide d'*«instructions»* qui constitue le *«programme»* exécuté par la machine. Les instructions sont au nombre de 4, notées : r_j`++`, r_j`--`, `ifz` r_j `goto` i', `halt`.
- un entier i qui représente un indice dans la suite d'instructions. On appelle cet indice le *«compteur ordinal»*.

L'*«exécution»* d'un programme par une machine à registres est définie par la *«fonction de transition»* τ sur les états des machines à registres.

Notation : si p est un programme, on note $\#p$ sa longueur ; si $i<\#p$, $p[i]$ est l'instruction numéro i de la suite p.

La valeur de $\tau(i,p,r_0,r_1,\ldots,r_m)$, avec $i<\#p$, est définie par :

1. si $p[i]$ est l'instruction `halt` :
$$\tau(i,p,r_0,r_1,\ldots,r_m)=r_0$$
2. si $p[i]$ est l'instruction r_j`++`, avec $j\leq m$:
$$\tau(i,p,r_0,r_1,\ldots,r_m)=\tau(i+1,p,r_0,r_1,\ldots,r_j+1,\ldots,r_m)$$
3. si $p[i]$ est l'instruction r_j`--`, avec $j\leq m$:
$$\tau(i,p,r_0,r_1,\ldots,r_m)=\tau(i+1,p,r_0,r_1,\ldots,r_j-1,\ldots,r_m)$$
4. si $p[i]$ est l'instruction `ifz` r_j `goto` i' avec $j\leq m$ et $i'<\#p$:
   - si $r_j=0$:
   $$\tau(i,p,r_0,r_1,\ldots,r_m)=\tau(i',p,r_0,r_1,\ldots,r_m)$$
   - sinon :
   $$\tau(i,p,r_0,r_1,\ldots,r_m)=\tau(i+1,p,r_0,r_1,\ldots,r_m)$$
5. sinon :
$$\tau(i,p,r_0,r_1,\ldots,r_m)=\tau(i,p,r_0,r_1,\ldots,r_m)$$

Quelques commentaires sur la fonction τ :

- on reconnaît dans les instructions r_j`++`, r_j`--`, `ifz` r_j `goto` i' et `halt` les opérations d'incrément, de décrément, de branchement conditionnel (si $r_j = 0$) et d'arrêt ;
- lorsqu'elle s'arrête, la machine livre, comme résulat du calcul, la valeur du registre r_0 ;
- la fonction τ n'est pas partout définie :
  - elle n'est pas définie si p ne termine pas par `halt` ;
  - elle n'est pas définie si $\#p \leq i$;
  - enfin, elle n'est pas définie si p est un programme qui *«boucle»*.

Une remarque sur l'instruction `halt` : pour être correctement exécuté et pouvoir produire une valeur, un programme doit contenir au moins une instruction `halt`. On peut, sans perte de généralité, faire l'hypothèse que dans tout programme p l'instruction `halt` est la dernière de la suite et que c'est là sa seule occurence dans p.

Notation on écrit les programmes comme une suite de ligne dont chacune contient une seule instruction. Pour rendre un peu plus lisibles les instructions de saut on utilise le biais syntaxique suivant : plutôt qu'un indice absolu, on indique un saut *«relatif»*. Par exemple `ifz` z_i `goto` $+3$, signifie un saut 3 lignes plus bas dans le programme ; `ifz` z_i `goto` -3, signifie un saut 3 lignes plus haut dans le programme.

Plus formellement :

si $p[i]$ est l'instruction `ifz` r_j `goto` d et
si $r_j = 0 : \tau(i, p, r_0, r_1, \ldots, r_m) = \tau(i + d, p, r_0, r_1, \ldots, r_m)$
sinon : $\tau(i, p, r_0, r_1, \ldots, r_m) = \tau(i + 1, p, r_0, r_1, \ldots, r_m)$

Exercice: montrer que pour tout programme qui utilise la syntaxe des sauts relatifs il existe un programme équivalent qui utilise l'instruction originelle de saut absolu.

Une autre astuce syntaxique consiste à utiliser une *«étiquette»* de saut. Dans ce cas, on fait figurer cette étiquettes devant l'instruction se trouvant à la ligne visée (on sépare l'étiquette de l'instruction par le caractère *«deux points»*).

Dans tout programme qui utilise des étiquettes de saut, on peut remplacer celles-ci par un indice absolu. On définit pour cela la fonction idx telle que si p est un programme et a une étiquette dans p, $idx(a, p)$ donne l'indice de la (première) ligne d'étiquette a dans p (ie : il existe une instruction x telle que $p[i] = a : x$). Posons $idx(a, p) = idx^*(a, p, 0)$ avec

$$\begin{array}{lcll} idx^*(a,p,i) & = & i & \text{si } \exists x.p[i] = a : x \text{ ou } i \geq \#p \\ idx^*(a,p,i) & = & idx^*(a,p,i+1) & \text{sinon} \end{array}$$

Exercice: dire les conditions sous lesquelles pour tout programme qui utilise des étiquettes, il exsite un programme équivalent qui utilise des indices de saut relatifs et le montrer.

On appelle *«relativisé»* d'un programme le programme équivalent obtenu en remplaçant les étiquettes par des indices de saut relatifs.

Exemples le programme suivant ajoute à r_i la valeur de r_j, avec $i \neq j$ et en supposant que $r_k = 0$:

`ifz` r_j `goto` $+4$
r_i`++`
r_j`--`
`ifz` r_k `goto` -3
`halt`

Même programme en utilisant des étiquettes :

a_0 : `ifz` r_j `goto` a_1
r_i`++`
r_j`--`
`ifz` r_k `goto` a_0
a_1 : `halt`

1.2.1 Macros instructions

Le jeu d'instructions des machines à registre donne un langage de programmation d'extrêmenent bas niveau. Nous allons l'enrichir de quelques *«macros instructions»* qui rendront la suite de l'exposition plus facile.

Saut inconditionnel la seule instruction de saut dont nous disposons est le saut conditionnel `ifz` r_i `goto` d. Pour obtenir un saut inconditionnel, il suffit de considérer un registre toujours nul. Convenons de l'appeler z. Cela est toujours possible puisque l'on dispose d'un nombre illimité de registres. On note $Goto(a)$ pour `ifz` z `goto` a, où a est soit un entier relatif, soit une étiquette.

Remise à zéro la suite d'instructions suivante a pour effet de (re)mettre la valeur de r_i à 0 :

`ifz` r_i `goto` $+3$
r_i`--`
$Goto(-2)$

On note $Raz(r_i)$ cette suite d'instructions.

Le saut en $+3$ lorsque $r_i = 0$ peut paraître surprenant, puisqu'il ne correspond à aucune instruction de la suite. Il se comprend si l'on songe que la macro Raz est appelée à être utilisée dans une suite d'instructions plus grande. Cette utilité sera illustrée par la suite.

Addition la suite d'instructions suivante (qui reprend l'exemple ci-dessus) a pour effet d'ajouter à r_i la valeur de r_j :

`ifz` r_j `goto` $+4$
r_i`++`
r_j`--`
$Goto(-3)$

Attention : cette suite d'instructions a également pour effet de mettre r_j à 0. Notons $Addz(r_i, r_j)$ cette suite.

Transfert de registre on veut une macro Sto à deux paramètres r_i et r_j telle que $Sto(r_i, r_j)$ a pour effet de donner au registre r_i la valeur du registre r_j. Pour ce faire, à l'instar de la macro $Goto$, on se donne un registre t qui sera dédié aux transferts. Voici comment l'on opère :

$Raz(r_i)$
`ifz` r_j `goto` $+5$
r_i`++`
t`++`
r_j`--`
$Goto(-4)$
`ifz` t `goto` $+4$
r_j`++`
t`--`
$Goto(-3)$

Dans un repmier temps, la macro *«copie»* r_j dans r_i *et* t (registre *t*ampon), puis reftransfert t dans r_j. La macro est correcte si t vaut 0 au moment de son invocation. Elle s'achève avec un t égal à 0.

Remarquons que l'on y voit l'utilité du saut *«une ligne trop bas»* des macros *Raz* et *Addz* pour enchaîner une remise à zéro ou une addition avec un autre traitement.

On appelle *«registres de service»* les registres distingués z et t. Par convention, on placera ces deux registres en fin de la liste des registres utilisés par un programme. On convient également que nous n'utiliserons jamais explicitement ces registres dans nos programmes.

1.2.2 Fonctions, calcul et programmes

Soient f une fonction d'arité k et p un programme, on dit que *«p calcule f (avec ρ registres)»* si $\rho = 1 + k + n$, pour un certain n et

- pour tout $x_1, \ldots, x_k$, où $f(x_1, \ldots, x_k)$ est défini, avec $r_0 = r_{k+1} = \ldots = r_{k+n} = z = t = 0$, on a

$$\tau(0, p, r_0, x_1, \ldots, x_k, r_{k+1}, \ldots, r_{k+n}, z, t) = f(x_1, \ldots, x_k)$$

- et pour tout $x_1, \ldots, x_k$, où $f(x_1, \ldots, x_k)$ n'est pas défini, avec $r_0 = r_{k+1} = \ldots = r_{k+n} = z = t = 0$,

$$\tau(0, p, r_0, x_1, \ldots, x_k, r_{k+1}, \ldots, r_{k+n}, z, t) \text{ n'est pas défini.}$$

Remarquons que pour calculer la valeur de $f(x_1, \ldots, x_k)$, l'appel à la fonction d'exécution τ *«initialise»* les registres $r_1, \ldots, r_k$ avec les valeurs respectives de $x_1, \ldots, x_k$ et les registres $r_0, r_{k+1}, \ldots, r_{k+n}, z, t$ avec la valeur 0.

On dit d'une fonction f qu'elle *«programmable»* lorsqu'il existe un programme p qui calcule f.

Voici quelques exemples de programmes qui *«calculent»* des fonctions simples.

L'addition la fonction d'addition est calculée par le programme suivant :

$Addz(r_0, r_1)$
$Addz(r_0, r_2)$
`halt`

Notons que ce programme a également pour effet d'annuler les valeurs initiales de r_1 et r_2.

Multiplication la multiplication est définie par itération de l'addition :

$$\begin{array}{lcl} 0 \times x_2 & = & 0 \\ (x_1 + 1) \times x_2 & = & x_2 + (x_1 \times x_2) \end{array}$$

Le résultat du produit de x_1 par x_2 est obtenu en répétant x_1 fois l'addition de x_2.

Pour écrire un programme qui calcule la multiplication, on peut donc utiliser un programme qui calcule l'addition. Dans les faits, on n'utilisera pas le programme lui-même, mais son relativisé privé de sa dernière ligne (qui contient l'instruction `halt`). Pour l'addition, appelons P_+ la suite d'instructions ainsi obtenue. L'exécution (des lignes) de P_+ a pour effet de mettre dans r_0 la somme de r_1 et r_2. La suite d'instructions de P_+ est une *macro* pour l'addition.

Voici comment utiliser P_+ pour calculer la multiplication :

$$\begin{array}{ll} & Sto(r_3, r_1) \\ & Sto(r_4, r_2) \\ a_0 : & \texttt{ifz } r_3 \texttt{ goto } a_1 \\ & Sto(r_1, r_0) \\ & Sto(r_2, r_4) \\ & P_+ \\ & r_3\texttt{--} \\ & Goto(a_0) \\ a_1 : & \texttt{halt} \end{array}$$

Notez comment les valeurs initiales de r_1 et r_2 sont *«sauvegardées»* dans r_3 et r_4 et comment les valeurs de r_0 et r_4 sont tranférées dans r_1 et r_2 avant l'exécution de P_+.

Exponentielle l'exponentiation est l'itération de la multiplication. Soit $P_\times$ la macro déduit du programme ci-dessus qui calcule la multiplication. En calquant le programme pour la multiplication, on obtient un programme qui calcule la fonction exponentielle $(r_1^{r_2})$.

$$\begin{array}{ll} & r_0\texttt{++} \\ & Sto(r_3, r_2) \\ & Sto(r_4, r_1) \\ a_0 : & \texttt{ifz } r_3 \texttt{ goto } a_1 \\ & Sto(r_1, r_0) \\ & Sto(r_2, r_4) \\ & P_\times \\ & r_3\texttt{--} \\ & Goto(a_0) \\ a_1 : & \texttt{halt} \end{array}$$

Notez que dans $P_\times$ l'indice relatif qui a remplacé l'étiquette a_1 provoque l'utile saut *«une ligne trop bas»* déjà remarqué.

Macrotisation de façon générale si f est programmable, on notera P_f la suite d'instructions d'un programme relativisé qui calcule f, privé de sa dernière instruction (`halt`). On parlera alors de *procédure qui calcule f* ou, plus simplement, de *procédure pour f*. Notez que les paramètres de telles procédures sont implicites : registres r_1, r_2, etc., pour les arguments ; registre r_0 pour le résultat. C'est au programme qui utilise de telles macros de veiller à l'initialisation des registres pour ganrantir l'exécution correcte des calculs.

2 Équivalence des modèles

2.1 Les fonctions récursives sont programmables

L'ensemble des fonctions récursives est fondé sur un ensemble de fonctions primitives (les constantes, le successeurs et les projections) que l'on enrichit en appliquant des mécanismes de constructions de nouvelles fonctions aux fonctions déjà existantes (la composition, le schéma primitif récursif, la minimisation) ; et toutes les fonctions récursives s'obtiennent ainsi. Pour montrer que toutes les fonction récursives sont programmables il suffit donc de montrer que les fonctions primitives sont programmables et que chaque mécanisme de construction de nouvelle fonction récursive est reproductible par programmes.

R1 les fonctions constantes $c(x_1, \ldots, x_k)$ sont calculées par le programme

```
halt
```

R2 la fonction successeur $s(x)$ est calculée par le programme

r_1++
$Sto(r_0, r_1)$
halt

R3 les projections $p_i(x_1, \ldots, x_k)$ avec $i \leq k$ sont calculées, pour chaque i, par le programme

$Sto(r_0, r_i)$
halt

R4 la composition $h(g_1(x_1, \ldots, x_k), \ldots, g_n(x_1, \ldots, x_k))$: par hypothèse, les fonctions h, g_1, ..., g_n sont programmables. On a donc des procédures $P_h, P_{g_1}, \ldots, P_{g_n}$ pour chacune de ces fonctions. Pour calculer la composition $h(g_1(x_1, \ldots, x_k), \ldots, g_n(x_1, \ldots, x_k))$, on exécute successivement $P_{g_1}, \ldots, P_{g_n}$ puis on exécute P_h. Le travail à faire est de déterminer et gérer correctement les registres nécessaires au calcul. Posons que l'exécution d'un P_{g_i} consomme $1+k+q_i$ registres (1 pour r_0, k pour $r_1, \ldots, r_k$ et q_i éventuels registres additionnels) et que l'exécution de P_h consomme $1 + n + q$ registres. Le programme pour la composition aura au moins besoin du maximum de ces valeurs : soit m ce nombre. Par définition, $k < m$ et $n < m$. Les registres au-delà de k (pour les calcul des g_i : $r_{k+1}, \ldots, r_m$) ou au-delà de n (pour le calcul de h : $r_{k+1}, \ldots, r_m$) sont dits *«auxiliaires»*.
À ces m registres, il convient d'en ajouter n pour stocker les résultats des calculs des P_{g_i} avant l'invocation de P_h.
De plus, rien ne garantit que l'exécution d'un P_{g_i} n'altère pas le contenu de l'un au moins des $r_1, \ldots, r_k$. Il convient donc de se donner k registres de *«sauvegarde»* dont on s'assure qu'ils ne seront pas altérés par les calculs des P_{g_i} et que l'on initialisera avec les valeurs $x_1, \ldots, x_k$ (i.e. celles de $r_1, \ldots, r_k$ à l'initialisation de l'exécution du programme).
Au total, on aura donc $m + n + k$ registres auxquels on adjoint les deux registres de service z et t. La suite des registres nécessaires au calcul de la composition se répartit donc ainsi

r_0 registre du résultat final

$r_1, \ldots, r_m$ registres des procédures P_{g_i} et P_h

$r_{m+1}, \ldots r_{m+n}$ registres pour les résultats intermédiaires

$r_{m+n+1}, \ldots, r_{m+n+k}$ registres de sauvegarde des paramètres initiaux

z, t registres de service

Pour alléger l'écriture, on note $MSto(r_i \ldots r_j; r'_i \ldots r'_j)$ la suite $Sto(r_i, r'_i) \ldots Sto(r_j, r'_j)$. On utilise le même genre d'abus de notation pour la macro Raz. En ajoutant quelques commentaires sur ce qui est fait, voici un programme qui calcule la composition $h(g_1(x_1, \ldots, x_k), \ldots, g_n(x_1, \ldots, x_k))$

$MSto(r_{m+n+1} \ldots r_{m+n+k}; r_1 \ldots r_k)$	*sauvegarde des valeurs de* $x_1, \ldots, x_k$
P_{g_1}	*calcul de* $g_1(x_1, \ldots, x_k)$
$Sto(r_{m+1}, r_0)$	*sauvegarde de la valeur de* $g_1(x_1, \ldots, x_k)$
$Raz(r_0)$	*remise à zéro de* r_0
$MSto(r_1 \ldots r_k; r_{m+n+1} \ldots r_{m+n+k})$	*(ré)initialisation des* $r_1, \ldots, r_k$ *avec* $x_1, \ldots x_k$
$MRaz(r_{k+1} \ldots r_m)$	*remise à zéro des registres auxiliaires*
$\vdots$	*calculs des* $P_{g_2}, \ldots, P_{g_{n-1}}$ *avec sauvegardes, remises à zéro et réinitialisations*
P_{g_n}	*calcul de* $g_n(x_1, \ldots, x_k)$
$Sto(r_{m+n}, r_0)$	*sauvegarde de la valeur de* $g_n(x_1, \ldots, x_k)$
$Raz(r_0)$	*remise à zéro de* r_0
$Sto(r_1 \ldots r_n, r_{m+1} \ldots r_{m+n})$	*initialisation des* $r_1, \ldots, r_n$ *avec les calculs intermédiaires*
$Raz(r_{n+1} \ldots r_m)$	*remise à zéro des registres auxiliaires pour* h
P_h	*calcul de* $h(g_1(x_1, \ldots, x_k), \ldots, g_n(x_1, \ldots, x_k))$
halt	

R5 le schéma de récurence primitive :

$$\begin{array}{lcl} f(0, x_1, \ldots, x_k) & = & g(x_1, \ldots, x_k) \\ f(x+1, x_1, \ldots, x_k) & = & h(x, x_1, \ldots, x_k, f(x, x_1, \ldots, x_k)) \end{array}$$

Si l'on *«déroule»* le calcul de f donné par les équations du schéma, on obtient

$$\begin{array}{lcl} f(0, x_1, \ldots, x_k) & = & g(x_1, \ldots, x_k) \\ f(1, x_1, \ldots, x_k) & = & h(0, x_1, \ldots, x_k, g(x_1, \ldots, x_k)) \\ & \vdots & \\ f(x, x_1, \ldots, x_k) & = & h(x-1, x_1, \ldots, x_k, \\ & & \quad h(x-2, x_1, \ldots, x_k, \\ & & \quad\quad \vdots \\ & & \quad\quad h(1, x_1, \ldots, x_k, \\ & & \quad\quad\quad h(0, x_1, \ldots, x_k, \\ & & \quad\quad\quad\quad g(x_1, \ldots, x_k))) \ldots)) \end{array}$$

Le calcul de $f(x, x_1, \ldots, x_k)$ se résume à une suite de compositions dont on obtient la valeur en calculant g, puis x fois h. C'est-à-dire : $g(x_1, \ldots, x_k)$ (i.e. $f(0, x_1, \ldots, x_k)$), qui donne une valeur v_0 ; puis $h(0, x_1, \ldots, x_k, v_0)$ (i.e. $f(1, x_1, \ldots, x_k)$), qui donne une valeur v_1 ; ... ; enfin, $h(x-1, x_1, \ldots, x_k, v_{x-1})$ (i.e. $f(x, x_1, \ldots, x_k)$), qui donne la valeur v_x.
Schématiquement, l'itérations se décompose en

$$\begin{array}{ll} & i = 0 \\ & v = g(x_1, \ldots, x_k) \\ a_1 : & if\ x = 0\ goto\ a_2 \\ & v = h(i, x_1, \ldots, x_k, v) \\ & x-- \\ & i++ \\ & goto\ a_1 \\ a_2 : & halt \end{array}$$

Puisqu'il s'agit d'une suite de compositions, on mettra en œuvre les mécanismes déjà vus de sauvegarde des résultats intermédiaires et de restauration des registres. Par hypothèse, les fonctions h et g sont programmables, soient P_h et P_g les procédures pour h et g. Soit m le nombre de registres maximal requis par P_h et P_g. On a $k+1 < m$. Les registres nécessaires au calcul se répartissent comme suit

r_0 registre du résultat

$r_1 \ldots r_{k+1}$ registres des paramètres initiaux

$r_{k+2} \ldots r_m$ (éventuels) registres additionnels

r_{m+1} registre pour les valeurs intermédiaires de l'itération (dernier argument de g)

r_{m+2} registre du compteur d'itérations (varie de x à 0)

r_{m+3} registre de l'itération, premier argument pour g, (varie de 0 à $x-1$)

$r_{m+4} \ldots r_{m+4+k}$ registres de sauvegarde des paramètres $x_1, \ldots x_k$

t, z registres de service

Un programme pour f est

$$
\begin{array}{ll}
 & Sto(r_{m+2}, r_1) \\
 & MSto(r_{m+4} \dots r_{m+4+k}; r_2 \dots r_{k+1}) \\
 & P_g \\
a_1 : & \texttt{ifz } r_{m+2} \texttt{ goto } a_2 \\
 & Sto(r_1, r_{m+3}) \\
 & MSto(r_2 \dots r_{k+1}; r_{m+4} \dots r_{m+4+k}) \\
 & Sto(r_{k+2}, r_0) \\
 & Raz(r_0) \\
 & MRaz(r_{k+3} \dots r_m) \\
 & P_h S \\
 & r_{m+2}\texttt{--} \\
 & r_{m+3}\texttt{++} \\
 & Goto(a_1) \\
a_2 : & \texttt{halt}
\end{array}
$$

R6 le schéma de minimisation $f(x_1, \dots, x_k) = \mu x.(g(x_1, \dots, x_k, x) = 0)$ suggère une méthode de calcul par *«force brute»* : essayer toutes les valeurs de x jusqu'à trouver la bonne. Ce que réalise le schéma de programme suivant :

$$
\begin{array}{ll}
 & x = 0 \\
a_1 : & v = g(x_1, \dots, x_n) \\
 & if\ v = 0\ goto\ a_2 \\
 & x\texttt{++} \\
 & goto\ a_1 \\
a_2 : & \texttt{halt}
\end{array}
$$

Par hypothèse, la fonction g est programmable. Soit P_g la procédure induite. On a (possiblement) l'itération de la procédure P_g, il faut donc prendre garde à la sauvegarde et à la restauration des valeurs des registres qui se répartissent en

r_0 registre pour le résultat final

$r_1 \dots r_{k+1}$ registres pour les arguments de g

$r_{k+2} \dots r_m$ registres additionnels pour le calcul de g

$r_{m+1} \dots r_{m+k}$ registres de sauvegarde des $x_1 \dots x_k$

r_{m+k+1} registre pour x (varie de 0 à ...)

z et t registres de service

Un programme pour réaliser le calcul du schéma de minimisation est :

$$
\begin{array}{ll}
 & MSto(r_{m+1} \dots r_{m+k}; r_1 \dots r_k) \\
a_1 : & P_g \\
 & \texttt{ifz } r_0 \texttt{ goto } a_2 \\
 & r_{m+k+1}\texttt{++} \\
 & Raz(r_0) \\
 & MSto(r_1 \dots r_k; r_{m+1} \dots r_{m+k}) \\
 & Sto(r_{k+1}, r_{m+k+1}) \\
 & MRaz(r_{k+2} \dots r_m) \\
 & Goto(a_1) \\
a_2 : & Sto(r_0, r_{m+k+1}) \\
 & \texttt{halt}
\end{array}
$$

Commentaire pour être complet, il faut montrer que les (schémas de) programmes donnés sont *corrects* ; c'est-à-dire que pour toute fonction f, si P_f est le programme construit selon la méthode proposée pour chacun des cas de définition de la fonction récursive f alors P_f calcule f.

2.2 Les programmes calculent des fonctions récursives

Ce qu'il faut montrer est que pour tout programme p, if existe une fonction récursive f telle que

$$\tau(0, p, 0, x_1, \ldots, x_k, 0, \ldots, \ldots) = f(x_1, \ldots, x_k)$$

L'idée de la démonstration est simplement de faire de τ une fonction récursive. Les fonctions récursives sont des fonctions sur les entiers (et non les programmes). On procède en deux étapes

1. on associe à chaque programme un (unique) entier que l'on appelle son *«code»*. Le code d'un programme doit est construit de telle façon que l'on soit capable d'en retrouver les constituants (les instruction et leurs arguments).
2. définir une fonction récursive qui réalise sur les entiers le travail spécifié par la fonction de transition τ.

2.2.1 Codage des programmes

Les paires On numérote les paires d'entiers (n, p) selon le principe suivant (dû à G. Cantor) : les paires d'entiers sont rangées dans un tableau à deux dimensions que l'on parcourt diagonale par diagonale. On numérote les paires selon l'ordre de ce parcours. Le schéma ci-dessous illustre ce principe :

$$\begin{array}{ccccccccc}
(0,0) & \rightarrow & (0,1) & \rightarrow & (0,2) & \rightarrow & (0,3) & \ldots \\
 & \swarrow & & \swarrow & & \swarrow & & \\
(1,0) & & (1,1) & & (1,2) & & \ldots & \\
 & \swarrow & & \swarrow & & & & \\
(2,0) & & (2,1) & & \ldots & & & \\
\end{array}$$

Pour calculer le rang de la paire (n, p) dans cette énumération, on se donne la fonction α telle que

$$\alpha(n, p) = (\Sigma_{i=0}^{n+p} i) + n = \frac{(n + p + 1)(n + p)}{2} + n$$

On notera $\langle n, p \rangle$ pour $\alpha(n, p)$.

On remarque que $n \leq \langle n, p \rangle$ et $p \leq \langle n, p \rangle$. Les fonctions inverses de α sont les deux *«projections»* fst et snd telles que $fst(\langle n, p \rangle) = n$ et $snd(\langle n, p \rangle) = p$ sont définies par minimisation et existentielle bornées :

$$fst(c) = \mu n \leq c.\exists p \leq c.(\langle n, p \rangle = c)$$

$$snd(c) = \mu p \leq c.\exists n \leq c.(\langle n, p \rangle = c)$$

Les listes finies (d'entiers) sont des structures définies par les valeurs et opérations suivantes :

- la liste vide (notée $[]$) ;
- une opération d'ajout d'un élément x en tête d'une liste xs (notée $x :: xs$) ;
- une opération d'extraction du premier élément d'une liste xs (notée $hd(xs)$) ;
- une opération d'extraction de la suite d'une liste xs (i.e. xs privée de son premier élément, notée $tl(xs)$).

On pose :

$[] = 0$
$x :: xs = \langle x, xs \rangle + 1$
$hd(xs) = fst(xs - 1)$
$tl(xs) = snd(xs - 1)$

On ajoute 1 à la fonction de construction des paires de telle sorte que la liste $0 :: []$ vaille 1 et non pas 0 qui représente la liste vide. On a $hd([]) = 0$ qui est un résultat ambigu puisque qu'aussi bien $hd(0 :: xs) = 0$; il faut donc prendre garde à l'utilisation de la fonction hd. On a $tl([]) = 0 = [] = tl(x :: [])$ ce qui est également ambigu.

Schéma de récurence sur les listes. Soient g et h deux fonctions primitives récusives. Soit f telle que

$$\begin{array}{lcl} f([], y) & = & g(y) \\ f(x :: xs, y) & = & h(x, xs, y, f(xs, y)) \end{array}$$

Exercice: montrer que f est primitive récursive.

Les programmes les 4 instructions des machines à registres sont codées de la façon suivante :

r_j`++`	$\langle 0, j\rangle$
r_j`--`	$\langle 1, j\rangle$
`ifz` r_j `goto` i'	$\langle 2, \langle j, i'\rangle\rangle$
`halt`	$\langle 3, 0\rangle$

Les programmes sont des listes d'instructions. On a donc une fonction qui à chaque programme associe un entier propre. La fonction de codage est injective, mais n'est pas surjective (i.e. certains entiers ne sont pas des codes de programmes) ce qui ne sera pas gênant pour notre propos.

Remarque: la fonction de codage des programmes est *primitive récursive*.

2.2.2 Fonction d'exécution d'un programme

Notre but est donc d'obtenir une version récursive (au sens de 1.1) de la fonction τ définie en 1.2.

On appelle *«état»* d'un programme p la paire (i, rs) où i est un numéro de ligne dans p et rs la liste des registres utilisés par p. La fonction de transition τ, telle que formulée en 1.2 n'est pas immédiatement récursive (au sens de 1.1). Nous allons donc définir une fonction récursive $exec$ telle que si $\hat{p}$ est le code d'un programme p, $exec(\hat{p}, x_1, \ldots, x_k) = \tau(0, p, 0, x_1, \ldots, x_k, 0, \ldots 0)$.

Il est un peu compliqué de montrer qu'un schéma de définition tel que celui donné pour τ est celui d'une fonction récursive. Pour définir $exec$, on a encore une fois recours à la *«force brute»* : partant d'un état initial, on calcule la liste des états d'une exécution du programme et, une fois cette liste construite, on va chercher dans son dernier élément (qui correspond à celui obtenu par l'exécution de l'instruction `halt`) la valeur résultant du calcul (contenu du registre r_0).

Pour atteindre ce but, un bon nombre de fonctions intermédiaires et utilitaires vont être nécessaires.

Incrément et décrément d'un registre

$incrs(j, rs)$: incrémente le j-ième élément de la liste (de registres) rs.

$$\begin{array}{lcl} incrs(j, []) & = & [] \\ incrs(0, r :: rs) & = & (r+1) :: rs \\ incrs(j+1, r :: rs) & = & r :: incrs(j, rs) \end{array}$$

$decrs(j, rs)$: décrémente le j-ième élément de la liste (de registres) rs.

$$\begin{array}{lcl} decrs(j, []) & = & [] \\ decrs(0, r :: rs) & = & (r-1) :: rs \\ decrs(j+1, r :: rs) & = & r :: decrs(j, rs) \end{array}$$

Remarque: un indice j hors de la liste rs laisse rs inchangé.

Utilitaires pour les listes

$nth(i, xs)$: donne le i-ème élément de la liste xs.

$$\begin{array}{lcl} nth(i, []) & = & 0 \\ nth(0, x :: xs) & = & x \\ nth(i+1, x :: xs) & = & nth(i, xs) \end{array}$$

Attention: un indice i hors de la liste xs donne la valeur 0 qui peut être ambigüe.

$len(xs)$: donne la longueur (i.e. le nombre d'éléments) de la liste xs.

$$\begin{array}{lcl} len([]) & = & 0 \\ len(x :: xs) & = & 1 + len(xs) \end{array}$$

$lst(xs)$: donne le dernier élément de xs.

$$lst(xs) \quad = \quad nth(len(xs) - 1, xs)$$

Attention: $lst([])$ vaut 0 qui peut être une valeur ambigüe.

$append(xs, ys)$: fonction de concaténation des listes.

$$\begin{array}{lcl} append([], ys) & = & ys \\ append(x :: xs, ys) & = & x :: append(xs, ys) \end{array}$$

On notera $xs@ys$ pour $append(xs, ys)$.

Reconnaissance et décomposition des instructions on se donne un ensemble d'utilitaires permettant de reconnaître dans un entier le codage d'une instruction de machine à registres ainsi que l'accès à ces arguments (registre, indice de saut).

$$\begin{array}{lcl} isInc(x) & = & (fst(x) = 0) \\ isDec(x) & = & (fst(x) = 1) \\ isGoto(x) & = & (fst(x) = 2) \\ isHalt(x) & = & (fst(x) = 3) \end{array}$$

$$\begin{array}{lcl} incArg(x) & = & snd(x) \\ decArg(x) & = & snd(x) \\ gotoArg_1 & = & fst(snd(x)) \\ gotoArg_2 & = & snd(snd(x)) \end{array}$$

Étape(s) d'exécution

$step(c, i, rs)$: calcule (le code de) l'état résultant de l'instruction c lorsque le compteur ordinal vaut i et les registres rs.

$$\begin{array}{lcll} step(c, i, rs) & = & \langle i+1, incrs(incArg(c), rs)\rangle & \text{si } isInc(c) \\ & = & \langle i-1, decrs(decArg(c), rs)\rangle & \text{si } isDec(c) \\ & = & if(gotoArg_1(c) = 0, \langle gotoArg_2(c), rs\rangle, \langle i+1, rs\rangle) & \text{si } isGoto(c) \\ & = & \langle i, rs\rangle & \text{sinon} \end{array}$$

Le dernier cas inclut l'instruction `halt` et les entiers c qui ne codent pas une instruction.

$isStep(p, i, rs, e)$: teste si une machine fait passer de l'état (i, rs) à l'état (codé par) e en exécutant $p[i]$.

$$isStep(p, i, rs, e) \quad = \quad (step(nth(i, p), i, rs) = e)$$

$isExec(p, es)$: teste si es est une suite d'états valides pour une exécution de p.

$$\begin{array}{lcl} isExec(p, []) & = & 0 \\ isExec(p, e :: []) & = & 1 \\ isExec(p, e_1 :: e_2 :: []) & = & isStep(p, fst(e_1), snd(e_1), e_2) \\ & & \wedge\ isExec(p, e_2 :: es) \end{array}$$

Initialisation Nous avons définit en 1.2.2 qu'un programme calcule une fonction d'arguments $x_1, \ldots, x_k$ lorsque ses registres $r_1, \ldots, r_k$ ont pour valeurs initiales $x_1, \ldots, x_k$, et tous ses autres registres ont pour valeur 0. On définit ici une fonction qui donnera l'état initial d'un programme pour calculer une fonction de $x_1, \ldots, x_k$.

On a besoin pour ce faire de connaître le nombre de registres utilisés par un programme : $m(p)$ donne ce nombre.

$$\begin{array}{lcll} m([]) & = & 0 & \\ m(c :: p) & = & max(incArg(c), m(p)) & \text{si } isInc(c) \\ & = & max(decArg(c), m(p)) & \text{si } isDec(c) \\ & = & max(gotoArg_1(c), m(p)) & \text{si } isGoto(c) \\ & = & m(p) & \text{sinon} \end{array}$$

$init_0(x)$: calcule une liste de longueur x ne contenant que des 0.

$$\begin{array}{lcl} init_0(0) & = & [] \\ init_0(x+1) & = & 0 :: init_0(x) \end{array}$$

$init(p, x_1, \ldots, x_k)$: donne l'état initial de p pour le calcul d'une fonction de $x_1, \ldots, x_k$.

$$init(p, x_1, \ldots, x_k) \quad = \quad \langle 0, 0 :: x_1 :: \ldots :: x_k :: init_0(m(p) - k) \rangle$$

Fonction d'exécution

$execTrace(p, x_1, \ldots, x_k)$: donne la liste des états obtenus par l'exécution de p pour les valeurs initiales $x_1, \ldots, x_k$. Cette liste est appelée *«trace»* de l'exécution de p. On demande qu'elle vérifie que l'état initial est celui attendu (voir fonction *init*), que la suite des états corresponde à des étapes valides (voir fonction *step*) et que le compteur ordinal de l'état final (le dernier de la liste) corresponde à l'instruction `halt` dans p.

$$\begin{array}{lcl} execTrace(p, x_1, \ldots, x_k) & = & \mu es.(hd(z) = init(p, x_1, \ldots, x_k) \\ & & \quad \wedge\ isExec(p, es) \\ & & \quad \wedge\ isHalt(nth(fst(lst(z)), p))) \end{array}$$

Jusqu'ici, les fonctions définies étaient primitives récursives. Cette dernière, définie par minimisation, est simplement récursive. Elle peut ne pas avoir de valeur pour certains p ou certains $x_1, \ldots, x_k$: soit que p n'est pas le code d'un programme, soit que p est le code d'un programme qui *«boucle»* sur les *«entrées»* $x_1 \ldots, x_k$. Aucune de ces situations ne nous gêne puisque, le premier cas est hors de notre propos (on ne cherche à savoir que ce que calculent les programmes), le second cas correspond à un programme qui calcule une fonction *«partielle»* (non partout définie).

$exec(p, x_1, \ldots, x_k)$: donne (si elle existe) la valeur de *«la fonction calculée par»* p avec les arguments $x_1, \ldots, x_k$ (i.e. valeur du registre r_0 de l'état terminal).

$$exec(p, x_1, \ldots, x_k) \quad = \quad fst(snd(lst(execTrace(p, x_1, \ldots, x_k))))$$

La fonction *exec* est récursive : notre but est atteint.

Commentaire pour être complet, il faut montrer la *correction* de la fonction *exec*; c'est-à-dire que pour tout programme p, si $\hat{p}$ est son code alors

$$\tau(0, p, 0, x_1, \ldots, x_n, 0, \ldots, 0) = exec(\hat{p}, x_1, \ldots, x_n)$$

3 Le problème de l'arrêt, son indécidabilité

Le *«problème de l'arrêt»* que nous envisageons ici est celui de l'*«arrêt»* (i.e. terminaison de l'exécution) des programmes des machines à registres. Un programme *«s'arrête»* lorsque son exécution atteint l'instruction `halt` au bout d'un nombre fini de transitions (un *«temps»* fini).

Le problème de l'arrêt d'un programme est un vrai problème dans la mesure où il existe effectivement des programmes qui *«bouclent»*, ne s'arrêtent pas :

```
0 : ifz z goto 0
1 : halt
```

Souvenons nous que, par convention, le registre de service z a toujours pour valeur 0 ; donc, dans le programme ci-dessus, l'instruction `halt` n'est jamais atteinte. D'autre part, l'exécution d'un programme dépend de ses *«entrées»*. Par exemple :

```
0 : ifz r1 goto 0
1 : halt
```

boucle si r_1 a pour valeur initiale 0 est s'arrête (i.e. atteint l'instruction `halt`) sinon. Le problème de l'arrêt (des programmes) dépend donc de deux paramètres : un programme et ses données initiales.

En résumé, le problème de l'arrêt (des programmes) est celui de savoir si oui ou non un programme p, quelconque, s'arrête quelles que soient ses données initiales. C'est un *«problème de décision»* : sa solution (sa réponse) est *«oui»* ou *«non»* (de façon plus générale : *«vrai»* ou *«faux»* ; *«0»* ou *«1»*). Une chose est de poser un problème, une autre est de le résoudre. Les méthodes pour ce faire peuvent varier. Pour ce qui est des programmes, on peut procéder par tests et décider qu'un programme ne boucle pas après un nombre donné d'essais positifs. Une autre méthode envisageable, puisque les programmes sont équivalentes à des données numériques, est l'utilisation d'un procédé de calcul de la réponse. On dit qu'un problème de décision est *«décidable»* s'il existe un procédé *«calculable»* permettant de le résoudre.

Tirons donc parti du fait que nous savons représenter les programmes par des entiers et du fait que nous savons coder un n-uplet d'entiers $(x_1, \ldots, x_n)$ par un seul entier x, pour formuler ainsi le problème de la décidabilité du problème de l'arrêt (des programmes) :

> *existe-t-il une fonction récursive A telle que pour tout (code de) programme p et pour toute donnée initiale x,*
>
> $$\begin{array}{rcll} A(p,x) & = & 1 & \text{si } exec(p,x) \text{ s'arrête} \\ & = & 0 & \text{sinon} \end{array}$$

La réponse à cette question est non : il n'existe pas de telle fonction. En effet, supposons qu'au contraire, A existe. C'est, par hypothèse une fonction récursive *«totale»* (i.e. définie pour toutes valeurs de ses arguments). À partir de A, on peut définir

$$\begin{array}{rcll} B(x) & = & exec(x,x)+1 & \text{si } A(x,x)=1 \\ & = & 0 & \text{sinon} \end{array}$$

On obtient alors la contradiction suivante : si A est récursive totale, B également ; en tant que fonction récursive, B est codable par un entier, disons b ; puisque B est totale, on a en particulier que $exec(b,b)$ est défini et donc $A(b,b)=1$; d'où $B(b)=exec(b,b)+1$; mais comme, par définition de $exec$, $B(b)=exec(b,b)$, on obtient $B(b)=B(b)+1$ qui est la contradiction recherchée.

4 Moralité

Quelqu'intelligent que soit votre compilateur (qui est une fonction programmable, donc récursive), il ne pourra vous garantir à tous les coups si votre programme boucle ou non.

Table des matières