Calculabilité

Pascal Manoury

8 novembre 2012

*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

Ces modèles visent à capturer le cœur de la notion de calculabilité. Ils ne se veulent pas réalistes en ce sens que

- ils ne concernent que les fonctions des entiers naturels dans les entiers naturels – nous verrons toutefois que l'on peut déjà y définir et manipuler des structures de données ;
- ils ne s'intéressent pas encore à la notion d'efficacité ni d'effectivité en temps ou en espace mais seulement à la notion de possibilité d'un processus de calculs.

Leur principe est de poser un ensemble minimal de briques de départ et de moyens de composition ou combinaison.

1.1 Fonctions récursives

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

R0 la constante 0

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

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

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 g et h deux fonctions récursives, la fonction f telle que

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

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 $g(x_1, \ldots, x_k, x) = 0$. 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.
Pour l'intuition informatique : les primitives récursives correspondent à des itérations bornées (boucles `for`) et le schéma μ à des itérations non bornées (boucles `while`).

Les entiers «*formels*» sont tous les termes que l'on peut écrire avec la constante 0 et la fonction s. Ils sont en bijection avec les entiers «*naturels*» (ensemble $\boldsymbol{N}$) :

Entiers	
naturels	formels
0	0
1	$s(0)$
2	$s(s(0))$
3	$s(s(s(0)))$
etc.	
en général et en abrégé	
n	$s^n(0)$

Si n est un entier naturel (*i.e.* $n \in \boldsymbol{N}$) on écrit $\underline{n}$ pour l'entier formel correspondant (*i.e.* $s^n(0)$).
On a sur les entiers formels le principe de raisonnement par induction suivant : pour toute formule $\Phi(x)$,
- si $\Phi(0)$ est vraie
- et si pour tout x, $\Phi(x) \Rightarrow \Phi(s(x))$
- alors pour tout $n \in \boldsymbol{N}$, $\Phi(\underline{n})$ est vraie.

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 dès lors qu'il existe une fonction f' définie selon les clauses R1-R5 telle que pour tout $n \in \boldsymbol{N}$, $f'(\underline{n}) = f(\underline{n})$.

> Le critère d'appartenance d'une fonction à la classe des primitives récursives est un *critère syntaxique*.

Nous donnons dans ce qui suit les définitions de quelques fonctions de base et montrons pourquoi elles sont primitives récursives. Et nous donnons également quelques exemples de *schémas de définitions* par récurrence d'usage courant et nous montrons qu'ils restent dans la classe des primitivs récursives.

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

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

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

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

qui répondent exactement au schéma primitif récursif.

La multiplication est primitive récursive. La multiplication est définie par

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

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

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

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

et est primitive récursive : on prend $g = 0$ et $h = p_1$ (on «*oublie*» l'appel récursif).

La fonction de soustraction *entière* ($0 - n = 0$) est obtenue par itération du prédécesseur.

$$\begin{array}{lcl} sub(x,0) & = & x \\ sub(x,s(y)) & = & pred(sub(x,y)) \end{array}$$

Elle est primitive récursive : on prend $g = p_1$ et $h(x,y,z) = pred(p_3(x,y,z))$ qui est primitive récursive par R4.

Commentaire: *stricto sensu*, il eût fallu montrer que définir par récurrence sur le second argument est aussi primitif récursif. Nous laissons cela en exercice.

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 usuelles ;
- par abus, on notera $x + 1$ pour $s(x)$;
- on s'autorisera également l'usage de 1 à la place de $s(0)$, etc. ;
- on ne mentionnera plus les usages triviaux des fonctions constantes, des projection ou de la composition dans les schémas de définition.

Fonctions booléennes, comparaisons 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

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

$$\begin{array}{lcl} and(x_1,x_2) & = & x_1 \cdot x_2 \end{array}$$

$$\begin{array}{lcl} or(x_1,x_2) & = & x_1 + x_2 \end{array}$$

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 *not* peut également être utilisée comme «*test à 0*» :

$$eqz(x) = not(x)$$

Les relation d'égalité (eq) et d'infériorité (lt) peuvent être définies de cette manière :

$$eq(x, y) = and(eqz(x - y), eqz(y - x))$$

$$lt(x, y) = and(eqz(x - y), not(eqz(y - x)))$$

On peut donc généralement définir les comparaisons usuelles en arithmétique $=, \neq, <, \leq, >$ et $\geq$ comme des fonctions primitives récursives et montrer la *correction* de ces définitions (exercice).

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

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

Elles est primitive récursive.

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 raccourci 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 $if(t_1, x_1, if(t_2, x_2, if(t_3, x_3, \ldots)\ldots)$.

On déduit des tests d'inégalité et de l'alternative 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 $x = y$.

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 $f^n(x) = f^*(n, x)$.

Remarque: **attention**, nous n'avons pas défini ici *une* fonction qui calculerait l'itérée d'une primitive récursive. Nous avons montré que *pour chaque fonction récursive f, nous savons définir une fonction f^* telle que $f^*(n,x)$ calcule $f^n(x)$.*

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 $P(x,y) = f(x) \cdot y$ avec f primitive récursive quelconque.

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

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

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

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

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.

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

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

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

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

qui est primitive récursive, et on a que pour tout $n, m \in N$, $f(\underline{n},\underline{m}) = f'(\underline{n},\underline{n},\underline{m})$ (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'

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

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 (*i.e.* > 0), 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 ...).

Où l'on voit que l'on ne cherche pas ici l'efficacité des calculs, mais leur justesse. Il eût en effet été plus économique de définir une fonction qui cherche le bon z en partant de 0 (exercice).

1.1.2 Paires et listes

Avec les seuls entiers naturels, il est possible de *représenter* des structures de données comme les paires et les listes. On parle de «*codage*» des structures.

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 $snd(\langle n, p\rangle) = p$. On les définit 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.

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. Le jeu d'instructions est réduit au strict minimum.

L'«*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 un registre : r_0) ;

– une suite finie non vide d'«*instructions*» qui constitue le «*programme*» exécuté par la machine. Les instructions sont au nombre de 3, notées : r_j`++`, r_j`--`, `ifz` r_j `goto` d.
– 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 $\tau(i, p, r_0, r_1, \ldots, r_m)$ est définie par les équations suivantes :

1. si $p[i]$ est l'instruction r_j`++`, avec $j \leq m$:
$$\tau(i, p, r_0, r_1, \ldots, r_m) = \tau(i+1, p, r_0, r_1, \ldots, r_j + 1, \ldots, r_m)$$
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 `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 :
   $$\tau(i, p, r_0, r_1, \ldots, r_m) = \tau(i+1, p, r_0, r_1, \ldots, r_m)$$
4. sinon :
$$\tau(i, p, r_0, r_1, \ldots, r_m) = r_0$$

Quelques commentaires sur la fonction τ :
– on reconnaît dans les instructions r_j`++`, r_j`--` et `ifz` r_j `goto` i' les opérations d'incrément, de décrément et de branchement conditionnel (lorsque $r_j = 0$) ;
– la machine s'arrête lorsque la valeur du compteur ordinal dépasse l'indice de la dernière instruction ou qu'elle rencontre une instruction inconnue ou une opération sur un registre inconnu ;
– lorsqu'elle s'arrête, la machine livre, comme résultat du calcul, la valeur du registre r_0 ;
– la fonction τ n'est pas partout définie : il existe des programmes qui «*bouclent*».

Stricto censu, il faudrait définir une «*famille*» de fonction de transitions τ_m pour chaque entier m, nombre de registres de la machine considérée.

Indices et étiquettes Voici un programme qui ajoute à r_i la valeur de r_j, avec $i \neq j$ et en supposant que $r_k = 0$:

0 : `ifz` r_j `goto` 4
1 : r_i`++`
2 : r_j`--`
3 : `ifz` r_k `goto` 0

Les chiffres précédents les instructions correspondent aux indices des dites instructions. Notez que l'indice (ou «*adresse*») de saut de la première instruction provoque l'arrêt du programme.

Les instructions de saut sont définies avec des indices *absolus*. Ce qui est laborieux à manipuler en pratique. Nous nous faciliterons la vie en remplaçant ces indices par des *étiquettes*, comme il est d'usage dans les langages de programmation. Ainsi, le petit exemple ci-dessus s'écrira

a_0 : `ifz` r_j `goto` a_x
r_i`++`
r_j`--`
`ifz` r_k `goto` a_0

avec cette convention que a_x représentera toujours l'indice de la dernière ligne + 1 (ici : 4).

Exercice: écrire la résolution des étiquettes.

1.2.1 Macros instructions

Le jeu d'instructions des machines à registre donne un langage de programmation d'extrêmement 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` a. 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.

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

a_0 : `ifz` r_i `goto` a_x
r_i`--`
$Goto(a_0)$

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

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 $t$ampon t qui sera dédié aux transferts. Voici comment l'on opère :

$Raz(t)$
$Raz(r_i)$
a_0 : `ifz` r_j `goto` a_1
r_i`++`
t`++`
r_j`--`
$Goto(a_0)$
a_1 `ifz` t `goto` a_x
r_j`++`
t`--`
$Goto(a_1)$

Aprés remise à zéro de r_i, la macro «*copie*» r_j dans r_i *et* t , puis t est retransféré dans r_j, de façon à restaurer la valeur de r_j altérée par la première boucle. La macro s'achève avec un t remis à 0.

Remarque: les indices ou étiquettes de saut compliquent un peu l'opération d'usage des macros. Pour être tout à fait correct, il faudrait procéder à un réajustement ou un renommage des indices ou étiquettes. Bien qu'une définition détaillée de cette opération ne soit pas immédiate, nous n'insisterons pas sur ce point et admettrons qu'une telle opération est toujours possible.

Nous engageons le lecteur à imaginer en exercice l'algorithme requis. Ces problèmes sont connus en informatique et correspondent aux algorithmes d'édition de liens et de relogement de code (le *loader*).

Sur les registres On appelle «*registres de service*» les registres distingués z et t. On convient que nous n'utiliserons jamais explicitement ces registres dans nos programmes : ils ne seront utilisés exclusivement par les macros Raz et Sto. Une machine mettra toujours en œuvres au moins les trois registres r_0, z et t. Dans une suite (ou *vecteur*) de registres, nous distinguerons donc

– le registre de *sortie* r_0 ;
– les registres de services z et t ;
– k registres *d'entrées* $r_1 \dots r_k$;
– n registres *auxiliaires* pour y stocker les valeurs intermédiaires nécessaires au calcul.

Un vecteur de registre aura donc la forme

$$r_0, r_1, \dots, r_k, r_{k+1}, \dots, r_{k+n}, z, t$$

1.2.2 Fonctions, calcul et programmes

Soient f une fonction d'arité k et p un programme, on dit que $\ll p$ *calcule* $f \gg$ si pour un n donné, on a
– pour tout $x_1, \dots, x_k$ tel que $f(x_1, \dots, x_k)$ est défini et pour $r_0 = r_{k+1} = \dots = r_{k+n} = z = t = 0$

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

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

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

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

On dit d'une fonction f qu'elle $\ll$*programmable*$\gg$ lorsqu'il existe un programme p qui calcule f.

Voici quelques exemples de programmes qui $\ll$*calculent*$\gg$ des fonctions simples.

Addition La fonction d'addition est calculée par le programme suivant :

$$\begin{array}{ll} & Sto(r_0, r_1) \\ a_0 & \texttt{ifz } r_2 \texttt{ goto } a_x \\ & r_0\texttt{++} \\ & r_2\texttt{--} \\ & Goto(a_0) \end{array}$$

Notez que ce programme annule la valeur de r_2. Appelons $PAdd$ ce programme.

Exercice: montrer la correction de ce programme, c'est-à-dire que

$$\tau(Padd, 0, x_1, x_2, 0, 0) = x_1 + x2$$

Multiplication La fonction de 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.

On peut schématiquement exprimer cette itération de la manière suivante :

$$\begin{array}{ll} a_1 : & if\ (r_2 = 0)\ goto\ a_x \\ & r_0 := r_0 + r_1 \\ & r_2\texttt{--} \\ & goto\ a_1 \\ a_x : & halt \end{array}$$

Pour écrire un programme qui calcule la multiplication, on peut donc utiliser un programme qui calcule l'addition. Par exemple, notre $PAdd$ dont l'exécution donne dans r_0 la valeur de $r_1 + r_2$. Il faudra faire un peu plus compliqué que notre schéma de façon à utiliser correctement les registres.

$$
\begin{array}{ll}
 & Raz(r_0) \\
 & Sto(r_3, r_1) \\
 & Sto(r_4, r_2) \\
a_0: & \texttt{ifz } r_3 \texttt{ goto } a_x \\
 & Sto(r_1, r_0) \\
 & Sto(r_2, r_4) \\
 & PAdd \\
 & r_3\texttt{--} \\
 & Goto(a_0)
\end{array}
$$

La valeur de r_0 est remise à zéro, les valeurs initiales de r_1 et r_2 sont tranférées dans r_3 et r_4. Commence alors la boucle de calcul qui travaillera à partir de r_4 qui reste constant (c'est le x_2 de la définition récursive) et de r_3 qui est décrémenté à chaque itération (c'est le x_1 de la définition récursive). Le registre r_0 sert d'≪*accumulateur*≫ lors du calcul : il est transféré dans r_1 et r_4 est transféré dans r_2 ; l'activation de $PAdd$ revient à faire que r_0 reçoit $r_0 + r_4$.

Remarque: naturellement, l'inclusion de $PAdd$ dans la séquence d'instructions pour la multiplication réclame que les noms d'étiquettes soient adaptés, en remplaçant a_0 par a_1 et a_x par a_2 que l'on rajouterait également comme étiquette de la ligne r_3--. Mais, dans la mesure où il peut toujours être fait, nous laisserons ce renommage implicite.

Pour programmer la multiplication, on aurait pu faire plus court, mais nous avons voulu montrer ici comment utiliser un bout de code, un programme, déjà existant ($PAdd$). On y retrouve des éléments du codage de bas niveau des appels de procédures dans les langages de programmation.

Exponentiiation L'exponentiation est l'itération de la multiplication. Soit $PMul$ le programme ci-dessus qui calcule la multiplication. En calquant le programme pour la multiplication, on obtient un programme qui calcule la fonction exponentielle ($r_1^{r_2}$).

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

≪***Macrotisation***≫ de façon générale si f est programmable, on notera P_f la suite d'instructions d'un programme qui calcule f. 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 garantir l'exécution correcte des calculs.

Pour prendre une analogie informatique, notre ≪*macrotisation*≫ revient à recourir systématiquement à *l'expansion de code* (*inlining*) lorsqu'une fonction ou procédure est invoquée – avec la ajustement nécessaires d'étiquettes.

Les vrais langages de programmation ont raffiné notre barbare notion de ≪*macrotisation*≫ en introduisant la précieuse notion de *procédure* et de *fonction* qui permet de partager et de réutiliser des portions de code correspondant aux calculs des ces fonctions ou procédures. Mais nous n'aurons pas besoin d'aller jusque là.

Instruction d'arrêt bien qu'elle ne soit pas absolument nécessaire (*cf* condition d'arrêt de la fonction de transition) il sera pratique d'avoir une instruction explicite d'arrêt que nous noterons `halt`. On peut alors compléter chacun des exemples donnés ci-dessus avec une dernière ligne :

$$a_x : \quad \texttt{halt}$$

Nous ferons toujours en sorte que nos programmes se terminent par une telle ligne et que ce soit là la seule occurence de l'instruction `halt`.

2 Équivalence des modèles

Montrer l'équivalence de nos deux modèles de la calculabilité, c'est montrer la validité des deux énoncés :

1. *Les fonctions récursives sont programmables.*
2. *Les programmes calculent des fonctions récursives.*

Nous ne ferons pas ici les preuves *in extenso* de ces énoncés. Nous nous contenterons

1. de définir ce qu'est un programme qui calcule une fonction et d'en donner un procédé général de construction.
2. de montrer comment **une** fonction récursive est capable de calculer le résultat de l'exécution d'un programme quelconque.

Nous pensons que les procédés que nous donnerons sont suffisamment clairs et intuitifs pour convaincre le lecteur qu'achever la preuve ne serait qu'une question d'un peu de temps et de travail.

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 (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 dans les programmes.

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

```
halt
```

ou, si l'on veut être sûr de son coup :

$Raz(r_0)$
`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 $h(g_1(x_1,\ldots,x_k),\ldots,g_n(x_1,\ldots,x_k))$: le calcul du résultat de la composition de fonctions $h, g_1, \ldots g_n$ se décompose schématiquement en

$$\begin{aligned} v_1 &:= g_1(x_1,\ldots,x_k)\\ &\vdots\\ v_n &:= g_n(x_1,\ldots,x_k)\\ r_0 &:= h(v_1,\ldots,v_n) \end{aligned}$$

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 réaliser la composition, il suffit donc de mettre en séquence les programmes $P_{g_1}, \ldots, P_{g_n}, P_h$ en omettant les dernière instruction (`halt`). Mais il faut également 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 $h : r_{n+1},\ldots,r_m$) 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 (les $v_1,\ldots,v_n$ de notre schéma).
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'itération* qui effectue ce calcul peut s'écrire

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

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 $k + 2 \leq m$. Les registres nécessaires au calcul se répartissent comme suit

r_0 registre du résultat

$r_1 \ldots r_m$ registres requis par P_h et p_g

r_{m+1} registre pour les valeurs intermédiaires de l'itération : le v de notre schéma, nous le noterons r_v

r_{m+2} registre du compteur d'itérations : le x de la récurrence ; nous le noterons r_x

r_{m+3} registre de l'itération : le i de notre schéma ; nousle noterons r_i

$r_{m+4} \ldots r_{m+4+k}$ registres de sauvegarde des paramètres $x_1, \ldots x_k$; nous les noterons $s_1, \ldots, s_k$

t, z registres de service

Un programme pour f est

$$
\begin{array}{rl}
 & Sto(r_x, r_1) \\
 & MSto(s_1 \dots s_k; r_2 \dots r_{k+1}) \\
 & MSto(r_1 \dots r_k; s_1 \dots s_k) \\
 & P_g \\
 & Sto(r_v, r_0) \\
 & Raz(r_i) \\
a_1 : & \texttt{ifz } r_x \texttt{ goto } a_2 \\
 & Sto(r_1, r_i) \\
 & MSto(r_2 \dots r_{k+1}; s_1 \dots s_k) \\
 & Sto(r_{k+2}, r_v) \\
 & Raz(r_0) \\
 & MRaz(r_{k+3} \dots r_m) \\
 & P_h \\
 & Sto(r_v, r_0) \\
 & r_x\texttt{--} \\
 & r_i\texttt{++} \\
 & Goto(a_1) \\
a_2 : & \texttt{halt}
\end{array}
$$

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 :

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

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 $x_1 \dots x_k$, notés $s_1, \dots, s_k$

r_{m+k+1} registre pour la valeur du x cherché, noté r_x

z, t registres de service

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

$$
\begin{array}{rl}
 & MSto(s_1 \dots s_k; r_1 \dots r_k) \\
 & Raz(r_x) \\
 & Sto(r_{k+1}, r_x) \\
a_1 : & P_g \\
 & \texttt{ifz } r_0 \texttt{ goto } a_2 \\
 & MSto(r_1 \dots r_k; s_1 \dots s_k) \\
 & r_x\texttt{++} \\
 & Sto(r_{k+1}, r_x) \\
 & Goto(a_1) \\
a_2 : & Sto(r_0, r_x) \\
 & \texttt{halt}
\end{array}
$$

Commentaire: pour être complet, il faudrait 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.

Sous l'habillage logico-mathématique de la *traduction* des constructions *de haut niveau* du langage des fonctions récursives vers les instructions *de bas niveau* des machines à registres , l'informaticien aura reconnu le processus qui lui est famillier de la *compilation*. La preuve de correction de la traduction ne serait rien d'autre que la preuve de correction d'un compilateur.

2.2 Les programmes calculent des fonctions récursives

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

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

La fonction de transition τ est à l'évidence calculable. L'idée de la démonstration est simplement d'en faire une *fonction récursive*. Ce qui nous donnerait immédiatement la fonction f cherchée.

Mais pour celà, nous devons surmonter deux obstacles :

1. les fonctions récursives sont des fonctions sur les entiers et non sur les programmes.
2. les clauses de la définition de τ sont très éloignées des schémas de définition des fonctions récursives.

Pour répondre à la première difficulté, nous allons montrer comment on peut associer à chaque suite d'instructions, et donc à chaque programme, un unique entier que l'on appelle son «*code*», construit de telle façon que l'on sera capable de retrouver ses constituants (les instruction et leurs arguments).

Pour répondre à la seconde difficulté, nous allons définir une fonction récursive qui, étant donné le code d'un programme et ses entrées, calcule un entier qui représente, non pas directement le résultat de l'exécution, mais la suite des états de la machine au cours de l'exécution du programme. C'est-à-dire, la *trace* de son exécution. Si le programme boucle, la trace est non définie ; sinon, nous saurons extraire de la trace produite le résultat de l'exécution. C'est-à-dire, le contenu du registre r_0 donné par l'état final.

2.2.1 Codage des programmes

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

Nous voulons donc obtenir une fonction récursive *exec* telle que si $\hat{p}$ est le code d'un programme p,

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

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).

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

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

Étape(s) d'exécution L'état d'exécution d'un programme est le couple constitué de la valeur du compteur ordinal et la valeur du vecteur de registres.

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

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

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

La fonction $isExec(p, es)$ 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 **??** 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 les entrées $x_1, \ldots, x_k$.

On a besoin pour ce faire de connaître le nombre de registres utilisés par un programme : la fonction $m(p)$ 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 $x_1, \ldots, x_k$.

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

Trace La fonction $execTrace(p, x_1, \ldots, x_k)$ donne la liste des états obtenus par l'exécution de p pour les valeurs initiales $x_1, \ldots, x_k$. Cette liste est appelée la 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.

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

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).

Commentaire: aucun informaticien raisonnable n'aurait implanter de cette manière la fonction de calcul de la trace de l'exécution d'un programme. C'est un peu comme si l'on programmait le calcul du quotient de x par y en énumérant tous les couples d'entiers (q, r) jusqu'à tomber sur celui qui vérifie $x = q \cdot y + r$. Mais rappelons nous qu'ici, nous ne sommes pas des ≪informaticiens raisonnables ≫, mais des logiciens dont le but est de montrer que l'on peut produire la trace de l'exécution d'un programme avec les seuls moyens offerts par le langage des fonctions récursives.

Exécution Enfin, la fonction $exec(p, x_1, \ldots, x_k)$ 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

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

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` r_1 `goto` 0
1 : `halt`

boucle si r_1 a pour valeur initiale 0 et 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 équivalents 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 $B(b)=exec(b,b)+1$, par définition de B. Mais comme, par définition de $exec$, $B(b)=exec(b,b)$, on aurait $B(b)=B(b)+1$; ce 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