Machines à registres

Paul Rozière
Paris7 – M2 LMFI

20 octobre 2005

1 Fonctions calculables par machines.

1.1 Définitions, programmes goto.

On va formaliser une notion de machine théorique très simple, les machines à registres. Ces machines ont une mémoire constituée d'un nombre fini de registres $R_0, R_1, \ldots, R_k$, chaque registre est de taille non bornée et peut donc contenir un entier arbitraire. Elles disposent d'autre part d'un programme qui est une suite finie constituée de 4 types d'instruction, et d'un index de lecture qui indique l'instruction du programme à exécuter. C'est un entier inférieur à la longueur du programme, les instructions étant numérotées de 1 en 1 à partir de 0.

1. incrémenter de 1 le registre numéro i, passer à l'instruction suivante :

$$R_i := R_i + 1$$

2. décrémenter de 1 le registre numéro i, passer à l'instruction suivante :

$$R_i := R_i - 1$$

3. exécuter un goto conditionnel, si le registre numéro i est nul, aller à l'instruction numéro p :

$$\mathsf{if}\ R_i = 0\ \mathsf{goto}\ p$$

4. une instruction d'arrêt qui apparait une et une seule fois en fin du programme.

$$\mathsf{halt}$$

On considère que la numérotation des instructions est implicite : le numéro désigne la place de l'instruction dans le programme, même si dans les exemples, on l'explicitera pour la clarté. Une telle suite d'instructions sera appelé *programme* goto dans la suite. Le calcul d'une telle machine peut ne pas terminer, et l'instruction goto est la seule instruction susceptible de conduire le calcul à ne pas terminer.

Remarquez que l'on peut accéder directement au registre numéro i : cela modélise plus ou moins la mémoire RAM (random access memory), mémoire à accès aléatoire. Vous verrez une autre notion de machine, les machines de Turing qui utilisent une mémoire à accès séquentiel : une machine de Turing écrit sur un ruban, il faut $|i - j|$ étapes pour aller de la case i à la case j.

Ces machines restent toutefois très théoriques : la taille de chaque registre, le nombre de registres, et la taille du programme ne sont pas bornées. Il manque par ailleurs des instructions essentielles pour manipuler les adresses de registres, même si elles ne permettraient pas de calculer de nouvelles fonctions.

Définition 1.1 Une fonction partielle f de $\mathbb{N}^n \to \mathbb{N}$ est calculable par une machine M à k registres, signifie que quand on initialise la machine en affectant aux registres $R_1, \ldots, R_{\inf(n,k)}$, les entiers $x_1, \ldots, x_{\inf(n,k)}$, aux registres restant (s'il en existe) la valeur 0, l'index de lecture étant à 0 (sur la première instruction), alors la machine termine son calcul si et seulement si $f(x_1, \ldots, x_n) \downarrow$ et alors la machine termine avec la valeur $f(x_1, \ldots, x_n)$ dans le registre R_0.

On n'a pas supposé ci-dessus dans la définition que $n \leq k$, mais évidemment :

Lemme 1.2 *Si $f : \mathbb{N}^n \to \mathbb{N}$ est calculable par une machine à k registres, alors f est calculable par une machine à k registres avec $k \geq n$.*

Démonstration. En effet, si $k < n$, il suffit d'ajouter les registres manquant $R_{k+1}, \ldots, R_n$, qui ne seront pas modifiés lors du calcul. ■

Voyons quelques exemples.

La machine à un seul registre R_0 et dont le programme contient pour seule instruction halt calcule les fonctions constantes égales à 0, $\lambda x_1 \ldots x_n.0$.

$\boxed{R_0}$	0	*halt*

On calcule l'addition avec la machine à 4 registres dont le programme est le suivant :

$\boxed{R_0}$	$\boxed{R_1}$	$\boxed{R_2}$	$\boxed{R_3}$		
				0	if $R_1 = 0$ goto 4
				1	$R_1 := R_1 - 1$
				2	$R_0 := R_0 + 1$
				3	if $R_3 = 0$ goto 0
				4	if $R_2 = 0$ goto 8
				5	$R_2 := R_2 - 1$
				6	$R_0 := R_0 + 1$
				7	if $R_3 = 0$ goto 4
				8	halt

Le registre R_3 ne sert qu'à pouvoir écrire un goto inconditionnel, instruction que l'on pourrait donc employer, sachant qu'on peut la simuler en ajoutant un registre à la machine. On va dans un premier temps montrer que l'on peut ainsi simuler quelques nouvelles instructions utiles.

Exercice 1

1. Décrire des machines qui calculent les fonctions sg et $\overline{sg}$.
2. Décrire une machine qui termine en 0 sur 0, et qui ne termine pas pour tout autre entier.

1.2 De nouvelles instructions.

Proposition 1.3 *Si une fonction partielle de $\mathbb{N}^n \to \mathbb{N}$ est calculée par une machine à registres utilisant en plus des instructions usuelles l'une des instructions suivantes*

1. *le* goto *inconditionnel, aller à la ligne p :*

$$\text{goto } p$$

2. *le registre numéro i est mis à 0 et l'on passe à l'instruction suivante :*

$$R_i := 0$$

3. *l'assignation d'un entier, le registre numéro i reçoit le contenu du registre numéro j ($j \neq i$) et l'on passe à l'instruction suivante :*

$$R_i := R_j$$

alors elle est calculable par une machine à registres usuelle.

Démonstration. On construit à chaque fois une machine qui simule la nouvelle instruction.

1. goto p : on a vu qu'il suffisait d'ajouter un registre dont le contenu sera toujours nul, on suppose $n \leq k$ (lemme 1.2), on ajoute alors un registre R_{k+1} et l'instruction devient if $R_{k+1} = 0$ goto p.

2. $R_i := 0$: la machine conserve les mêmes registres. On suppose que l'instruction $R_i := 0$ est à la place j. On remplace l'instruction $R_i := 0$ par la suite d'instructions :

$$\begin{array}{rl} j & \text{if } R_i = 0 \text{ goto } j+3 \\ j+1 & R_i := R_i - 1 \\ j+2 & \text{goto } j \end{array}$$

et dans le reste du programme on décale de 2 tous les goto l pour $l > j$ (remplacés par goto $l+2$).

3. $R_i := R_j$: on suppose $n \leq k$ (lemme 1.2), on ajoute alors un registre R_{k+1}. On suppose que l'instruction $R_i := 0$ est à la place j. On remplace l'instruction $R_i := 0$ par la suite d'instructions :

$$\begin{array}{rl} j & R_i := 0 \\ j+1 & \text{if } R_j = 0 \text{ goto } j+5 \\ j+2 & R_j := R_j - 1 \\ j+2 & R_i := R_i + 1 \\ j+3 & R_{k+1} := R_{k+1} + 1 \\ j+4 & \text{goto } j+1 \\ j+5 & \text{if } R_{k+1} = 0 \text{ goto } j+9 \\ j+6 & R_{k+1} := R_{k+1} - 1 \\ j+7 & R_i := R_i + 1 \\ j+8 & \text{goto } j+5 \end{array}$$

et dans le reste du programme on décale de 8 les goto l pour $l > j$. ■

Par exemple la projection p_n^i est calculée par une machine à n registres avec pour programme

$$\begin{array}{l} R_0 := R_i \\ \text{halt} \end{array}$$

donc par une machine à registres usuelle.

La fonction successeur est calculée par la machine à deux registres de programme :

$$\begin{array}{l} R_0 := R_1 \\ R_0 := R_0 + 1 \\ \text{halt} \end{array}$$

1.3 Programmes structurés.

On peut préferer utiliser des instructions plus complexes qui permettent de structurer les programmes.

Définition 1.4 Un *programme structuré* est une suite d'instructions, chaque instruction pouvant être (définition inductive) :

1. l'une des instructions d'assignation déjà décrite

$$R_i := 0,\ R_i := R_i + 1,\ R_i := R_i - 1,\ R_i := R_j\ (j \neq i)$$

2. une séquence d'instructions, elles sont exécutées séquentiellement, puis le programme passe à l'instruction suivante :

$$\text{begin } S_1;\ \ldots\ ; S_p \text{ end}$$

3. une instruction "while", une boucle qui répète une instruction S donnée

$$\text{while } R_i \neq 0 \text{ do } S$$

l'instruction S est exécutée tant que le contenu du registre numéro i n'est pas nul, s'il est nul on passe à l'instruction suivante.

4. une instruction "for", une boucle de répétition bornée d'une instruction S donnée

$$\mathsf{for}\ i = 1\ \mathsf{to}\ R_i\ \mathsf{do}\ S$$

L'instruction S est exécutée un nombre de fois égal à l'entier contenu dans le registre R_i *avant exécution de ces instructions*, puis l'on passe à l'instruction suivante. Même si le registre R_i est modifié par l'intruction S, le nombre de répétitions de l'instruction n'est pas modifié.

On appelle programme while les programmes structurés qui n'utilisent pas l'instruction for.

L'exécution d'une instruction est plus complexe que pour un programme goto : chaque instruction peut avoir en fait la même complexité qu'un programme. En particulier on peut avoir imbrication des while et des for. L'instruction halt est devenue inutile : les instructions du programme sont exécutées l'une après l'autre, mais la machine ne termine pas toujours son calcul pour autant, puisqu'une boucle while peut ne pas terminer. C'est d'ailleurs la seule instruction susceptible de conduire le programme à ne pas terminer.

Les fonctions partielles calculables sur machines à registre avec programme structuré ou programme while se définissent de la même façon qu'au paragraphe précédent, on suppose de plus que la machine a toujours au moins autant de registres que n l'arité de la fonction.

Lemme 1.5 *Si une fonction de $\mathbb{N}^n \to \mathbb{N}$ est calculée par une machine avec programme structuré, elle est calculable par une machine avec programme* while.

Démonstration. On a $n \geq k$. On procède par induction sur la définition des instructions (les boucles for peuvent être imbriquées). On montre le résultat en simulant chaque instruction d'un programme structuré sur une machine M à k registres par une instruction ou une séquence d'instruction sur une machine à $k+s$ registres, où s est le nombre d'instructions for dans le programme. À chaque instruction for est associé de façon univoque un registre R_{k+f}, $1 \leq f \leq s$. On suppose que S est simulée par une séquence d'instructions sans for $\mathcal{S}$, et on simule l'instruction

$$\mathsf{for}\ i = 1\ \mathsf{to}\ R_i\ \mathsf{do}\ S$$

par la séquence :

$R_{k+f} := R_i$
$\mathsf{while}\ R_{k+f} \neq 0\ \mathsf{do\ begin}\ R_{k+f} := R_{k+f} - 1;\ \mathcal{S}\ \mathsf{end}$

On doit bien recopier le registre R_i pour le laisser dans le même état à la fin du calcul. D'autre part la suite d'instructions $\mathcal{S}$ doit laisser le registre R_{k+f} dans le même état. ■

Proposition 1.6 *Si une fonction de $\mathbb{N}^n \to \mathbb{N}$ est calculée par une machine à registres avec programme structuré, elle est calculable par une machine à registres avec programme* goto.

Démonstration. D'après le lemme il suffit de le montre pour les programmes while. On le montre par induction. Il suffit de simuler chaque instruction d'un programme structuré par une séquence d'instructions avec goto. Le programme sera obtenu en juxtaposant dans l'ordre les séquences obtenues.

1. Les instructions d'assignations sont des instructions de base des machines à registre ou sont simulables d'après la proposition 1.3.
2. On suppose que les instructions$S_1, \ldots, S_p$ sont simulées par les suites d'instructions $\mathcal{S}_1, \ldots, \mathcal{S}_p$, appelons $\mathcal{S}$ la suite obtenue en les concaténant. Alors la séquence

$$\mathsf{begin}\ S_1; \ldots; S_p\ \mathsf{end}$$

est simulée par la suite d'instructions $\mathcal{S}$.

3. On suppose que l'instruction S est simulée par la suite d'instructions $\mathcal{S}$ de longueur s. Alors l'instruction

$$\mathsf{while}\ R_i \neq 0\ \mathsf{do}\ S$$

est simulée par la séquence (les lignes sont numérotées pour que ce soit plus clair) :

$$\begin{array}{rl} \vdots & \vdots \\ l & \mathsf{if}\ R_i = 0\ \mathsf{goto}\ l + (s+2) \\ \vdots & \mathcal{S} \\ l+s+1 & \mathsf{goto}\ l - (s+1) \\ l+s+2 & \ldots \\ \vdots & \vdots \end{array}$$

Les programmes structuré apparaissent donc d'une certaine façon comme des cas particuliers de programme goto. On va voir que les machines avec programmes goto, ou programmes structurés calculent en fait la même classe de fonction : les fonctions récursives. Pour cela le lemme suivant sera pratique :

Définition 1.7 Une machine est dite *propre* quand elle termine son calcul avec tous ses registres à l'exception du registre de sortie R_0 dans le même état qu'au début du calcul.

Lemme 1.8 *Si une fonctione est calculable par machine à registres avec programme structuré, alors elle est calculable par une machine propre (avec programme structuré).*

Démonstration. Il suffit de copier les registres (sauf celui de sortie), puis de calculer sur ces copies sans toucher aux registres initiaux. Soit M une machine à $k+1$ registres $R_0, \ldots, R_{k+1}$, dont le programme est une suite d'instructions $(S_1, \ldots, S_s)$ qui calcule f. On construit une machine propre à $2k+1$ registres $R_0, \ldots, R_k, R'_1, \ldots, R'_k$ ($R'_i = R_{i+k}$) de la façon suivante :

$$\begin{array}{rl} 1 & R'_1 := R_1 \\ \vdots & \vdots \\ k & R'_k := R_k \\ k+1 & S_1[R'_i/R_i] \\ \vdots & \vdots \\ k+s & S_s[R'_s/R_s] \end{array}$$

■

On ne se servira pas de ce lemme pour les machines avec programme goto, mais il reste valide : il faut alors décaler les goto de k dans les instructions supplémentaires.

Exercice 2

1. Simuler directement l'instruction for par un programme goto.
2. Simuler directement l'instruction

   $$\mathsf{repeat}\ S\ \mathsf{until}\ R_i = 0$$

   par un programme goto.

2 Les fonctions récursives sont calculables par machines.

On montre maintenant comment calculer n'importe quelle fonction récursive partielle par un programme structuré.

2.1 Les fonctions récursives partielles.

Proposition 2.1 *Les fonctions récursives partielles sont calculables par machines à registres avec programme structuré. Par conséquent les fonctions récursives sont calculables sur machines à registres avec programme* while *et programme* goto.

Démonstration. On procède par induction sur la définition des fonctions récursives partielles. On a déjà traité en exemple les fonctions de base, $\lambda x.0$, les projections p_k^i et la fonction successeur. On vérifie que les instruction utilisées sont bien celles des programmes structurés.

composition Supposons que les fonctions partielles $g_1, \ldots, g_k : \mathbb{N}^n \to \mathbb{N}$ et $h : \mathbb{N}^k \to \mathbb{N}$ sont calculables par des machines $M_1, \ldots, M_k$ et M_0, dont les programmes sont les suites d'instructions $\mathcal{S}_1, \ldots, \mathcal{S}_k$ et $\mathcal{S}_0$. On suppose ces machines propres d'après le lemme 1.8. On suppose que chacune de ces machines utilisent au plus $m+1$ registres, et que $m \geq n$ et $m \geq k$. On construit une machine M qui utilise $m+k+1$ registres que l'on va noter $R_0, R_1, \ldots, R_m, H_1, \ldots, H_k$. Elle va calculer successivement $g_1(x_1, \ldots, x_n)$,...,$g_k(x_1, \ldots, x_n)$ dont elle va stocker les valeurs dans les registres $H_1, \ldots, H_k$, puis calculer $f(H_1, \ldots, H_k)$. Voici le programme de cette machine

$\mathcal{S}_1$
$H_1 := R_0$
$R_0 := 0$
$\vdots$
$\mathcal{S}_k$
$H_k := R_0$
$R_0 := 0$
$R_1 := H_1$
$\vdots$
$R_k := H_k$
$R_{k+1} := 0$
$\vdots$
$R_m := 0$
$\mathcal{S}_0$

Remarquez que dès que l'une des machines $M_1, \ldots, M_k$ ne termine pas la machine M ne termine pas.

Récurrence primitive Soit M_1 une machine propre à k_1+1 registres dont le programme est $\mathcal{S}_1$ et qui calcule la fonction partielle $g : \mathbb{N}^n \to \mathbb{N}$ $(n \leq k_1 + 1)$. Soit M_2 une machine propre à k_2 registres dont le programme est $\mathcal{S}_2$ et qui calcule la fonction partielle $h : \mathbb{N}^{n+2} \to \mathbb{N}$ $(n+2 \leq k_2)$. Soit un entier m tel que $m \geq k_1$ et $m \geq k_2$. La fonction f est définie par récurrence primitive à partir de g et h :

$$f(\vec{a}, 0) = g(\vec{a})$$
$$f(\vec{a}, x+1) = h(\vec{a}, x, f(\vec{a}, x))\,.$$

La fonction f est calculée par une machine à $m+2$ registre dont voici les instructions :

$R_{m+1} := R_{n+1}$
$R_{n+1} := 0$
$\mathcal{S}_1$
for $i = 1$ to R_{m+1} do begin $R_{n+1} := R_{n+1} + 1$; $R_{n+2} := R_0$; $R_0 := 0$; $\mathcal{S}_2$ end

On calcule donc successivement $f(\vec{a}, 0)$ (avant la boucle for) puis dans l'ordre $f(\vec{a}, 1), \ldots, f(\vec{a}, n)$. Dès que l'une de ces valeurs n'est pas définie, la machine ne termine pas, ce qui est bien le comportement souhaité.

minimisation Soit M_0 une machine propre à $k+1$ registres dont le programme est $\mathcal{S}_0$ qui calcule la fonction partielle $g : \mathbb{N}^{n+1} \to \mathbb{N}$, avec $n+1 \leq k$. La fonction f :

$$f(x_1, \ldots, x_n) = \mu t.[g(x_1, \ldots, x_n, t) = 0]$$

est calculée par la machine à $k+1$ registres suivante :

$\mathcal{S}_0$
$R_0 := R_0 + 1$ while $R_0 \neq 0$ do begin $R_0 := 0$; $\mathcal{S}_0$; $R_{n+1} := R_{n+1} + 1$ end
$R_{n+1} := R_{n+1} - 1$
$R_0 := R_{n+1}$ ■

2.2 Les fonctions primitives récursives.

Si on examine la preuve, on se rend facilement compte que le while n'intervient que pour la minimisation. Le for suffit pour la récurrence primitive. Appelons programme for les programmes structurés qui n'utilisent pas de while, on a donc :

Proposition 2.2 *Les fonctions primitives récursives sont calculables par programme* for*.*

La réciproque de ce résultat est vraie.

Proposition 2.3 *Toute fonction calculable par programme* for *est primitive récursive.*

Il faut bien remarquer que les boucles for que nous avons défini ne peuvent modifier la variable d'itération lors du calcul, alors que c'est possible par exemple dans le langage C. Sans cette condition le résultat est manifestement faux (le programme ne termine pas forcément).

Exercice 3 Démontrer la proposition précédente : procéder par induction sur le niveau d'imbrication des boucles for. Montrer que le contenu de chaque registre de la machine est une fonction primitive récursive des entrées en utilisant le schéma de récurrences mutuelles (exercice 12 feuille 1).

3 Les fonctions calculables par machine sont récursives.

Nous allons maintenant de montrer que les fonctions partielles calculables par machine goto sont récursives. Pour cela on va coder par des entiers les machines elles-mêmes, et les états possibles d'une machine à un instant donné. Ensuite on montrera que le calcul de la machine sur un nombre t d'étapes, vue comme le calcul du code de son état au bout de t étapes, se code de façon récursive.

3.1 Machine, état d'un machine.

Une machine est déterminée par :

1. son nombre de registres ;

2. son programme qui est une suite finie d'instructions.

L'état d'un machine est déterminée par

1. un index de lecture (entier) : le numéro de l'instruction à exécuter ;

2. la suite finie des contenus des registres.

Le code d'une machine ou d'un état de machine est un entier, la fonction de codage doit être injective, mais pas nécessairement surjective. on va commencer par coder ses instructions.

3.1.1 Préliminaires et notations sur les fonctions primitives récursives.

On va utiliser le codage primitif récursif des couples et k-uplets défini à l'exercice 11 de la feuille 1 et celui des suites finies défini à l'exercice 13 de la feuille 1, ainsi que des fonctions définies dans ces exercices. Les arguments de la fonction $nth(l, i)$ qui calcule le i-ème élément de la suite de code l sont notés dans cet ordre.

Pour simplifier les notations on va noter ainsi les bijections de Cantor :

$$\alpha_k(x_1, \ldots, x_k) = \langle x_1, \ldots, x_k \rangle \ .$$

On rappelle que par définition des α_i :

$$\langle x_1, \langle x_2, \ldots, x_k \rangle \rangle = \langle x_1, \ldots, x_k \rangle \ .$$

et que les projections vérifient $\pi_2^1 = \pi_3^1 = \ldots$ et plus généralement :

$$\pi_n^i = \pi_m^i \text{ pour } i < n \leq m$$

Comme α_k est une bijection on obtient une définition correcte de fonction en écrivant :

$$f(x_1, \ldots, x_{i_1}, \langle y_1, \ldots, y_k \rangle, x_{i+1}, \ldots, x_n) = h(x_1, \ldots, x_{i_1}, y_1, \ldots, y_k, x_{i+1}, \ldots, x_n)$$

et si h est primitive récursive alors f est primitive récursive. En effet cette définition équivaut à :

$$f(x_1, \ldots, x_n) = h(x_1, \ldots, x_{i-1}, \pi_k^1(x_i), \ldots, \pi_k^k(x_i), x_{i+1}, x_n) \ .$$

3.1.2 Codage des instructions.

On a 4 sortes d'instructions, chacune pouvant être paramétrée par un entier (le numéro de registre pour l'incrémentation et la décrémentation), ou un couple d'entier (le numéro de registre et le numéro de ligne pour le goto conditionnel). On va donc coder chaque instruction par le code d'un couple, on note $\ulcorner S \urcorner$ l'entier qui code l'instruction S :

$$\begin{array}{ll} \ulcorner R_i := R_i + 1 \urcorner & = \langle 0, i \rangle \\ \ulcorner R_i := R_i - 1 \urcorner & = \langle 1, i \rangle \\ \ulcorner \mathsf{if}\ R_i = 0\ \mathsf{goto}\ n \urcorner & = \langle 2, \langle i, n \rangle \rangle (= \langle 2, i, n \rangle) \\ \ulcorner \mathsf{halt} \urcorner & = \langle 3, 0 \rangle \end{array}$$

De fait, le codage n'a pas besoin d'être fonctionnel, et nous considérerons dans la suite que tout entier c tel que ${\pi^2}_1(c) \geq 3$ est le code d'une instruction halt.

3.1.3 Codage des machines.

le code d'une machine à $k+1$ registres de programme $S_0, \ldots, S_s$ est l'entier :

$$\langle k, [\ulcorner S_1 \urcorner; \ldots; \ulcorner S_s \urcorner] \rangle \ .$$

Une machine possède au moins un registre, la première projection est donc le nombre de registres moins 1. Le codage est évidemment injectif, mais pas surjectif, ce qui n'est pas gênant. On n'aura même pas vraiment besoin du résultat de l'exercice suivant.

Exercice 4 Montrer que l'ensemble des entiers qui sont des codes de machine est primitif récursif.

3.1.4 Codage de l'état d'une machine.

On code l'état d'une machine dont les contenus des registres sont dans l'ordre $R_0, R_1, \ldots, R_k$ et l'index de lecture est i par l'entier :

$$\langle i, [R_0; R_1; \ldots; R_k] \rangle$$

3.2 Le calcul.

Supposons que la machine de code m prenne en entrée n entiers. Il nous faut essentiellement pour coder le calcul les deux fonctions et le prédicat suivants :

- Une fonction d'initialisation $init_n : \mathbb{N}^n \to \mathbb{N}$. Cette fonction calcule en fonction du code m d'une la machine et des entrée $x_1, \ldots, x_n$ l'état initial de cette machine (au départ du calcul).
- Une fonction de transition $tr : \mathbb{N}^2 \to \mathbb{N}$. Cette fonction calcule en fonction du code m d'une machine et de l'état e de cette machine l'état de la machine après une étape de calcul à partir de e.
- Un prédicat $Halt$ de terminaison à deux arguments qui est vrai pour les couples (m, e) où l'état codé par e indique que la machine codée par m est en fin de calcul.

Remarquez qu'il ne s'agit pas vraiment de définitions : on ne demande rien si les entrées ne sont pas cohérentes, ce qui est le cas par exemple pour ces fonctions et prédicat quand la première entrée n'est pas le code d'une machine, ou encore pour la fonction de transition et le prédicat de terminaison quand la seconde entrée n'est pas le code d'un état possible de la machine codée par la première entrée.

Il est intuitivement clair que l'on peut trouver de telles fonctions et prédicat calculables, donc récursifs d'après la thèse de Church. On montre explicitement dans les sections qui suivent que l'on peut trouver de telles fonctions et prédicat qui sont primitifs récursifs.

3.2.1 Fonction d'initialisation.

On définit d'abord une fonction auxiliaire qui crée en fonction de k la liste des $k+1$ premiers éléments de la suite infinie $0, x_1, \ldots, x_n, 0, \ldots$. Ensuite $init_n$ construit l'état initial de la machine de code m, à savoir un index de lecture à 0, et si celle-ci possède $k+1$ registres ($k = \pi_1^2(m)$), le code de la liste de longueur $k+1$ $[0; x_1; \ldots; x_n; 0; \ldots; 0]$ (liste éventuellement tronquée si $k < n$) :

$$
\begin{array}{l}
aux(x_1, \ldots, x_n, 0) = [0] = 0 :: [] \\
aux(x_1, \ldots, x_n, 1) = [0; x_1] = 0 :: x_1 :: [] \\
\vdots \\
aux(x_1, \ldots, x_n, n) = [0; x_1; \ldots; x_n] = 0 :: x_1 :: \ldots :: x_n :: [] \\
aux(x_1, \ldots, x_n, n+r+1) = aux(x_1, \ldots, x_n, n+r)@[0] \\
init_n(\langle k, s\rangle, x_1, \ldots, x_n) = \langle 1, aux(x_1, \ldots, x_n, k)\rangle
\end{array}
$$

Lemme 3.1 *la fonction $init_n(m, x_1, \ldots, x_n)$ définie ci-dessus est primitive récursive. Elle calcule, quand m est le code d'une machine, l'état initial de la machine de code m avec $x_1, \ldots, x_n$ en entrées.*

Démonstration. La fonction se comporte comme souhaité si m est le code d'une machine. Le schéma de récurrence utilisé est primitif récursif. En effet on le ramène à une récurrence primitive en utilisant dans l'étape de récurrence n fois le «si ... alors ... sinon ...» sur des conditions primitives récursives (singleton). Notez bien que n est une constante. ■

3.2.2 Fonction de transition.

On va avoir besoin pour écrire la fonction de transition de modifier le registre numéro i, ce qui demande, pour le codage considéré, de modifier le i-ème élément d'une liste.

Proposition 3.2 *la fonction inc, respectivement dec, qui appliquée à un entier i et au code d'une liste, incrémente le $i+1$-ème élément de la liste, respectivement décrémente ce $i+1$-ème élément est primitive récursive.*

Démonstration. On peut utiliser le schéma de récurrence avec substitution de paramètre (voir feuille 1 exercice 16), on a :

$$
\begin{array}{l}
\text{si } l = [] \text{ alors } inc(0, l) = [] \text{ sinon } inc(0, l) = (hd(l) + 1) :: tl(l) \\
inc(n+1, l) = hd(l) :: inc(n, tl(l))
\end{array}
$$

De même pour *dec*. ■

Par ailleurs on peut calculer de façon primitive récursive l'instruction courante : $inst(m, e)$ est le code de l'instruction de la machine de code $m = \langle n, s\rangle$ auquel renvoie l'index de lecture indiqué par l'état de code $e = \langle i, r\rangle$ (si l'index de lecture i est supérieur au nombre d'instructions, on considère qu'il renvoie à la dernière instruction) :

$$inst(\langle n, s\rangle, \langle i, r\rangle) = nth(s, inf(i, length(s))) \ .$$

On décompose maintenant la fonction de transition en ses deux projections, soient tr_1 et tr_2, et on montre que l'on peut définir chacune de façon primitive récursive.

$tr_1(m, e)$ calcule le numéro de l'instruction après exécution de l'instruction indiquée par $e = \langle i, r\rangle$:

$$
\begin{array}{l}
\text{si } \pi_2^1(inst(m, \langle i, r\rangle)) \in \{0, 1\} \qquad \text{(* si incrémentation ou décrémentation *)} \\
\quad \text{alors } tr_1(m, \langle i, r\rangle) = i + 1 \qquad \text{(* on passe à l'instruction suivante *)} \\
\quad \text{sinon si } \pi_2^1(inst(m, \langle i, r\rangle)) = 2 \qquad \text{(* si goto *)} \\
\qquad \text{alors si } nth(r, \pi_3^2(inst(m, \langle i, r\rangle))) = 0 \qquad \text{(* si } R_i = 0 \text{ *)} \\
\qquad\quad \text{alors } tr_1(m, \langle i, r\rangle) = \pi_3^3(inst(m, \langle i, r\rangle)) \text{ (* aller à la ligne indiquée *)} \\
\qquad\quad \text{sinon } tr_1(m, \langle i, r\rangle) = i + 1 \quad \text{(* on passe à l'instruction suivante *)} \\
\qquad \text{sinon } \ tr_1(m, \langle i, r\rangle) = i \qquad \text{(* } \pi_2^1(inst(m, \langle i, r\rangle)) > 2 \text{ : halt ou autre *)}
\end{array}
$$

$tr_2(m, e)$ calcule la liste des contenus des registres après exécution de l'instruction indiquée par $e = \langle i, r\rangle$:

si $\pi_2^1(inst(m, \langle i, r\rangle)) = 0$ (* si incrémentation *)
  alors $tr_2(m, \langle i, r\rangle) = inc(i, r)$ (* $R_i := R_i + 1$ *)
  sinon si $\pi_2^1(inst(m, \langle i, r\rangle)) = 1$ (* si décrémentation *)
    alors $tr_2(m, \langle i, r\rangle) = dec(i, r)$ (* $R_i := R_i - 1$ *)
    sinon $tr_2(m, \langle i, r\rangle) = r$ (* $\pi_2^1(inst(m, \langle i, r\rangle)) > 1$: goto, halt ou autre *)

Les deux fonctions tr_1 et tr_2 sont bien définies de façon primitive récursive, la fonction de transition

$$tr(m, e) = \langle tr_1(m, e), tr_2(m, e)\rangle$$

est donc primitive récursive. Résumons les résultats de cette section.

Lemme 3.3 *Les fonctions inst et tr définie ci-dessus sont primitives récursives. Quand m est le code d'une machine M et e le code d'un état E possible pour M, $inst(m, e)$ est le code de l'instruction courante du programme de M selon l'état E, $tr(m, e)$ est le code de l'état de M obtenu après exécution de cette instruction.*

3.2.3 Prédicat de terminaison.

On rappelle que l'état d'une machine est terminal si l'instruction indiquée par l'état est l'instruction halt. On peut donc définir le prédicat de terminaison $Halt$ ainsi :

$$Halt(m, e) \equiv \pi^1{}_2(inst(m, e)) > 2$$

Lemme 3.4 *Le prédicat binaire Halt défini ci-dessus est primitif récursif. Il est vrai pour les couples (m, e) où l'état codé par e indique que la machine codée par m est en fin de calcul, quand m est le code d'une machine, et e le code d'un état pour m.*

3.2.4 Les entiers qui ne sont pas des codes de machines.

Nous avons défini ci-dessus les fonctions d'initialisation et de transition et le prédicat de terminaison pour tous les entiers. Parmi ceux-ci certains ne codent pas de machine. Cela n'a pas d'importance : ces fonctions décrivent de toutes façon un comportement mécanique, et finalement nous allons montrer que les fonctions calculables par une classe plus étendue (au moins en apparence) de machines sont récursives partielles. Nous allons décrire leur comportement, ce qui ne sera vraiment utile qu'au paragraphe 3.4.2.

Pour un entier donné codant une machine, le nombre de registres est bien défini ainsi que le programme comme suite, il se peut simplement que des éléments de la suite ne correspondent pas à des instructions valides.

Nous avons déjà supposé (paragraphe 3.1.2) que toute entier c tel que $\pi^2{}_1(c) > 2$ codait une instruction halt, ce qui correspond bien au choix du prédicat de terminaison. La machine peut avoir des instructions halt n'importe où dans le programme, et a dernière instruction n'est pas forcément un halt, ce qui généralise un peu la notion initiale. La machine s'arrête dès qu'elle atteint un halt. Si la dernière instruction est exécutée et que ce n'est pas un halt ou un goto, alors cette intruction est répétée indéfiniment (voir définition de $inst$) : la machine ne s'arrête pas.

Les instructions d'incrémentation et de décrémentation ($\pi^2{}_1(c) = 0, 1$) pourraient concerner un numéro de registre qui n'apparait pas dans la machine. Dans ce cas on considère que l'instruction ne fait rien en dehors de passer à l'instruction suivante. C'est bien ce que code tr.

La condition de l'instruction goto peut porter sur un registre qui n'apparait pas dans la machine : on le considère comme un goto inconditionnel(voir définition de tr_1). Elle peut renvoyer à une ligne de programme d'index supérieur à la longueur du programme. Dans ce cas cela revient à renvoyer à la dernière ligne du programme (voir définition de $inst$).

3.2.5 Temps de calcul.

On a tout ce qu'il faut maintenant pour définir le temps de calcul, c'est à dire le nombre d'étapes jusqu'à ce que la machine termine, par minimisation.

On définit de façon primitive récursive la fonction $st_n(m, x_1, \ldots, x_n, t)$ qui calcule l'état de la machine de code m au bout de t étapes de calcul avec les entrées $x_1, \ldots, x_n$:

$$st_n(m, x_1, \ldots, x_n, 0) = init_n(m, x_1, \ldots, x_n)$$
$$st_n(m, x_1, \ldots, x_n, t+1) = tr(m, st_n(m, x_1, \ldots, x_n, t))$$

Enfin le prédicat de terminaison $H_n(m, x_1, \ldots, x_n, t)$, qui indique que la machine de code m avec en entrée $, x_1, \ldots, x_n$ s'est arrêtée au bout de t étapes de calcul, se définit de façon primitive récursive.

$$H_n((m, x_1, \ldots, x_n, t) \equiv Halt(m, st_n(m, x_1, \ldots, x_n, t))$$

Le *temps de calcul* de la machine de code m pour les entrées $x_1, \ldots, x_n$ est donc obtenu par minimisation, c'est, s'il existe :

$$\mu t.H_n(m, x_1, \ldots, x_n, t)$$

Étant donné un état e le contenu du registre R_0 est donné par $hd(\pi_2^2(e))$. La fonction calculée par la machine m est donc définie par :

$$hd \circ \pi_2^2 \circ st_n(m, x_1, \ldots, x_n, \mu t.H_n(m, x_1, \ldots, x_n, t))$$

donc est récursive partielle. Remarquez que la fonction partielle calculée est de toute façon récursive, que m soit ou non le code d'une machine. Résumons les résultats obtenus.

Proposition 3.5 *Pour toute entier $n \geq 1$, il existe une fonction primitive récursive $U_n : \mathbb{N}^{n+2} \to \mathbb{N}$, et un prédicat primitif récursif H_n à $n+2$ arguments tels que, si m est le code d'une machine M alors :*

- *$st_n(m, x_1, \ldots, x_n, t)$ est le code de l'état de la machine M avec en entrées $, x_1, \ldots, x_n$ au bout de t étapes de calcul ;*
- *$H_n(m, x_1, \ldots, x_n, t)$ ssi M avec en entrées $x_1, \ldots, x_n$ s'arrête au bout de t étapes de calcul.*
- *La fonction n-aire f calculée par M est récursive partielle et vérifie :*

$$f(x_1, \ldots, x_n) = U_n(m, x_1, \ldots, x_n, \mu t.H_n(m, x_1, \ldots, x_n, t))$$

Corollaire 3.6 *Toute fonction calculable par machine à registres avec programme* goto*, avec programme structuré, ou avec programme* while *est récursive partielle.*

Corollaire 3.7 *Toute fonction récursive partielle s'écrit comme composée d'une fonction primitive récursive et d'une fonction définie par minimisation sur un prédicat primitif récursif.*

On déduit du corollaire précédent que l'on obtient toutes les fonctions récursives partielles en restreignant dans la définition le schéma de minimisation $\mu t.[g(\vec{x}, t) = 0]$ aux fonctions g totales.

3.2.6 Temps de calcul d'une fonction primitive récursive.

Les fonctions primitives récursives définissent d'une certain façon une classe de complexité (très étendue !) comme le montrent les propositions qui suivent.

Proposition 3.8 *Si le temps de calcul (sur une machine à registres) d'une machine est borné par une fonction primitive récursive (en fonction de ses entrées) alors la fonction calculée par la machine est primitive récursive.*

Démonstration. On reprend la proposition 3.5. On peut borner la minimisation par une fonction primitive récursive, le prédicat H_n et la fonction U_n sont primitifs récursifs. ■

La réciproque de cette proposition est intuitivement vraie, mais plus pénible à démontrer.

Proposition 3.9 *Toute fonction primitive récursive est calculable par une machine dont le temps de calcul est une fonction primitive récursive des entrées.*

Démonstration. (indications). On montre d'abord que le temps de calcul d'une machine avec programme for est une fonction primitive récursive des entrées : c'est essentiellement le nombre de pas dans les boucles for, une succession de boucles for correspond à une addition, une imbrication à une multiplication. Le nombre de pas d'une boucle est le contenu d'un registre qui est une fonction primitive récursive des entrées (voir exercice 3). On montre ensuite que le temps de calcul reste une fonction primitive récursive par traduction d'un programme for en programme goto. ■

Évidemment une machine peut calculer une fonction primitive récursive sans que son temps de calcul soit une fonction primitive récursive des entrées ! Par exemple la fonction nulle est calculée par une machine obtenue en ajoutant en fin de programme l'instruction $R_i := 0$ pour n'importe quelle machine calculant une fonction totale.

3.3 Forme normale de Kleene.

On va donner une démonstration alternative des résultats précédents qui fournit une forme un peu plus simple pour les fonctions calculables (on va préciser le corollaire 3.7). On fait intervenir le code de la suite des états de la machine, plutôt que le temps de calcul (qui est la longueur de cette suite).

Proposition 3.10 *Pour chaque entier $n \geq 1$ il existe un prédicat primitif récursif notée T^n et une fonction primitive récursive $U : \mathbb{N} \to \mathbb{N}$ tels que pour toute machine de de code m calculant une fonction à n arguments f :*

1. $\exists s\ T^n(m, x_1, \ldots, x_n, s) \Leftrightarrow f(x_1, \ldots, x_n) \downarrow$
2. $T^n(m, x_1, \ldots, x_n, s) \Rightarrow U(s) = f(x_1, \ldots, x_n)$
3. $(T^n(m, x_1, \ldots, x_n, s)$ et $T^n(m, x_1, \ldots, x_n, s')) \Rightarrow s = s'$

Démonstration. L'idée est que s désigne le code (en commençant par la fin) d'une suite d'états correcte pour m, soit $[e_n; \ldots; e_0]$. On commence par définir un prédicat P_n qui indique que la suite d'états est bien correcte (sans que le calcul soit forcément fini), pour une machine de code m avec $x_1, \ldots, x_n$ en entrées. Le prédicat P_n est défini par sa fonction caractéristique f :

$$\begin{array}{l} f(m, x_1, \ldots, x_n, []) = 1 \\ f(m, x_1, \ldots, x_n, [e]) = \delta(init_n(m, x_1, \ldots, x_n), e) \\ f(m, x_1, \ldots, x_n, e :: s) = f(m, x_1, \ldots, x_n, e) \cdot \delta(tr(hd(s)), e) \end{array}$$

la fonction f est définie par récurrence structurelle sur les listes, à partir de fonctions récursives primitives. on a vu qu'alors f est récursive primitive. Le prédicat P_n est donc primitif récursif. On définit ensuite

$$\begin{array}{l} T'^n(m, x_1, \ldots, x_n, s) \equiv P_n(m, x_1, \ldots, x_n, s) \text{ et } Halt(m, hd(s)) \\ T^n(m, x_1, \ldots, x_n, s) \equiv T'^n(m, x_1, \ldots, x_n, s) \text{ et } \forall s' < s \neg T'^n(m, x_1, \ldots, x_n, s') \end{array}$$

Le prédicat primitif récursif T'^n ne vérifie a priori que la première condition de la proposition (terminaison). Le prédicat T^n est primitif récursif et vérifie les deux premières conditions : terminaison et fonctionalité pour s en sortie.

La fonction U a besoin d'extraire la valeur de R_0 du code du premier état de s soit :

$$U(s) = hd(\pi_1^2(hd(s)))$$

fonction qui est bien primitive récursive. ■

Corollaire 3.11 (forme normale de Kleene) *Les prédicats T^n et la fonction U sont ceux de la proposition précédente. Pour toute fonction récursive $f : \mathbb{N}^n \to \mathbb{N}$, il existe au moins un entier m tel que :*

$$f(x_1, \ldots, x_n) = U(\mu s.T^n(m, x_1, \ldots, x_n, s)) \ .$$

Démonstration. Il suffit de prendre le code m d'une machine qui calcule f. ■

Une expression de la fonction f comme celle donnée dans le corollaire :

$$f(x_1, \ldots, x_n) = g(\mu s.P(x_1, \ldots, x_n, s))$$

où $g : \mathbb{N} \to \mathbb{N}$ est une fonction primitive récursive et P un prédicat primitif récursif est dite sous *forme normale de Kleene.*

3.4 Fonctions universelles.

3.4.1 Propriété d'énumération.

Pour chaque $n \geq 1$ on définit la fonction d'arité $n+1$ φ^n :

$$\varphi^n(m, x_1, \ldots, x_n) = U(\mu s.T^n(m, x_1, \ldots, x_n, s))$$

et on note simplement φ pour φ^1.

Cette fonction récursive partielle permet de calculer toutes les fonctions récursives partielles à n arguments. On dit que c'est une *fonction universelle* pour les fonctions récursives partielles à n arguments. On peut réécrire ainsi le corollaire précédent :

Théorème 3.12 (Théorème d'énumération) *la famille de fonctions $(\varphi^n)_{n\in\mathbb{N}^*}$, définie ci-dessus est telle que pour toute fonction récursive partielle $f : \mathbb{N}^n \to \mathbb{N}$ il existe au moins un entier m tel que :*

$$f(x_1, \ldots, x_n) = \varphi^n(m, x_1, \ldots, x_n)$$

Un tel entier m sera appelé un *indice de f* (rappelons que c'est le code d'une machine qui calcule f). On notera $f = \varphi^n_m$ si m est un code de f.

En fait chacune des fonctions $(\varphi^n)_{n\in\mathbb{N}^*}$ est universelle en un sens un peu plus fort. Par exemple, la fonction $\varphi = \varphi^1$ énumère modulo codage toutes les fonctions récursives partielles :

Corollaire 3.13 *Pour toute fonction récursive partielle $f : \mathbb{N}^n \to \mathbb{N}$, il existe un entier i tel que :*

$$f(x_1, \ldots, x_n) = \varphi(i, \langle x_1, \ldots, x_n \rangle)$$

Démonstration. On prend pour i un indice de la fonction $\lambda x.f(\pi^n_1(x), \ldots, \pi^n_n(x))$. ■

Il est intuitivement clair (voir exercice suivant) qu'une fonction a une infinité d'indices : on peut toujours ajouter des détours inutiles dans le programme d'une machine. On montrera plus tard que c'est en un certain sens inévitable.

Exercice 5 (Lemme de rembourrage)

1. étant donné une machine M à k registres, construire une machine M', avec autant de registres, qui calcule la même fonction, et dont le programme a un code strictement plus grand que celui de M.
2. Montrer qu'il existe une fonction primitive récursive u_n telle que
$$\varphi^n_{u_n(m)} = \varphi^n_m \text{ et pour tout } m \ u_n(m) > m$$
3. Montrer qu'il existe une fonction primitive récursive $h_n(p, m)$ telle que h_p est strictement croissante en la première variable et pour tout entier m, $\varphi^n_{h(p,m)} = \varphi^n_m$.

Une conséquence immédiate du théorème d'énumération est que, la fonction φ^n étant récursive partielle à $n+1$ arguments, elle est a un indice : le code d'une machine qui la calcule, que l'on appelle *machine universelle.*

Corollaire 3.14 *Pour tout $n \geq 1$ il existe un entier m_n tel que :*

$$\varphi^n(m, x_1, \ldots, x_n) = \varphi^{n+1}(m_n, m, x_1, \ldots, x_n) \ .$$

3.4.2 Propriété de paramétrisation.

Une propriété en quelque sorte réciproque de celle que l'on vient d'énoncer, et caractéristique des familles de fonctions universelles, est la suivante.

Théorème 3.15 (théorème s_n^m) *Pour tous entiers $m, n \geq 1$ il existe une fonction récursive totale $s_n^m : \mathbb{N}^{n+1} \to \mathbb{N}$ telle que :*

$$\varphi^{m+n}(i, x_1, \ldots, x_n, y_1, \ldots, y_m) = \varphi^m(s_n^m(i, x_1, \ldots, x_n), y_1, \ldots, y_m)$$

Pour la famille de fonctions universelle étudiée ici, la fonction s_n^m est primitive récursive.

Si on on revient aux machines, la fonction s_n^m permet donc, à partir du code i d'une machine M à $m+n$ entrées, et de n entiers $x_1, \ldots, x_n$, de calculer le code d'une machine M' à m entrées, qui fonctionne comme M, après avoir fixer les n premières entrées à $x_1, \ldots, x_n$.

Démonstration. On va suivre la remarque précédente. Il s'agit d'abord d'indiquer comme construire M', puis on montre que les constructions se codent de façon primitive récursive. Il faut veiller à ce que la fonction calculée soit bien celle cherchée même quand l'indice $i = \langle k, l \rangle$ n'est pas un "vrai" code de machine (cf. paragraphe 3.2.4). La machine M' va avoir $sup(m+n, k)$ registres. On a construit son programme à partir de celui de M en 3 étapes.

1. On ajoute en tête du programme de M les instructions qui correspondent à :

$$R_1 := x_1, \ldots, R_n := x_n$$

chaque opération $R_i := x_i$ s'écrit :

$$\underbrace{R_i := R_i + 1, \ldots, R_i := R_i + 1}_{x_i}$$

2. On change dans le programme obtenu les numéros des registres (assignations et goto) :
- les registres de $n+1$ à $n+m$ sont décalés à partir de 1,
- les registres de 1 à n sont décalés à partir de $m+1$.
- les numéros de registre au delà de k sont décalés au delà de $sup(m+n, k)$: comme on ajoute éventuellement des registres il ne faut pas que des assignations ou des goto qui ne sont pas corrects (cf. paragraphe 3.2.4) le deviennent.

3. On décale les numéros de ligne n des goto n de $x_1 + \cdots + x_k$, pour tenir compte des instructions ajoutées en tête de programme.

Il est à peu près clair qu'à chaque étape correspond une opération primitive récursive sur le code. Ceux qui n'en sont pas convaincus sont invités à le démontrer eux-mêmes, ou, en dernier ressort, à chercher de possibles erreurs dans les détails donnés ci-dessous.

1. Le code d'une suite de x incrémentations sur le registre i se définit par récurrence primitive récursive sur x :

$$\begin{array}{l} a(i, x) = [] \\ a(i, x+1) = \langle 0, i \rangle :: a(i, x) \quad \text{(* 0 est le code de l'incrémentation *)} \end{array}$$

La fonction a est primitive récursive, et donc également la fonction $f_1(s, x_1, \ldots, x_n)$ qui ajoute la liste d'instructions souhaitée en tête du programme s (n est une constante) :

$$f_1(s, x_1, \ldots, x_n) = a(1, x_1) @ \ldots @ a(n, x_n) @ s$$

2 et 3. La fonction $num(j, k)$ qui calcule le nouveau numéro d'un registre i pour une machine à k registres est primitive récursive (n est une constante) :

$$\begin{array}{l} \text{si } 1 \leq j \leq n \text{ alors } num(j) = j + m \\ \quad \text{sinon si } n+1 \leq j \leq n+m \\ \quad\quad \text{alors } num(j) = j \dot{-} n \\ \quad\quad \text{sinon si } j > k \text{ alors } num(j) = j + sup(m+n, k) \\ \quad\quad\quad\quad \text{sinon } num(j) = j \end{array}$$

La fonction $d(k, g, ins)$ effectue le décalage souhaité du numéro de registre dans le code d'une instruction ins, et décale de g les goto. Elle est primitive récursive :

si $\pi_2^1(ins) \in \{0, 1\}$ (* incrémentation, décrémentation *)
  alors $d(k, g, ins) = \langle \pi_2^1(ins), num(\pi_2^2(ins), k) \rangle$
  sinon si $\pi_2^1(inst(m, \langle i, r \rangle)) = 2$ (* si goto *)
    alors $d(k, g, ins) = \langle 2, num(\pi_3^2(ins), k), \pi_3^3(ins) + g \rangle$
    sinon $d(k, g, ins) = ins$ (* $\pi_2^1(ins) > 2$: halt ou autre *)

Il suit que, par composition, la fonction $d'(i, x_1, \ldots, x_n, ins) = d((\pi_2^1(i), x_1 + \cdots + x_n, ins)$ est primitive récursive, et donc également La fonction $f_{2,3}(i, x_1, \ldots, x_n) = map_{d'}(\pi_2^2(i))$. Cette dernière construit à partir du code i de la machine M le code du programme avec les renumérotations de registres et décalages des goto souhaités.

On obtient finalement la fonction s_n^m de façon primitive récursive :

$$s_n^m(i, x_1, \ldots, x_n) = \langle sup(\pi_2^1(i), m + n), f_{2,3}(f_1(\pi^2{}_2(i), x_1, \ldots, x_n), x_1, \ldots, x_n) \rangle \quad \blacksquare$$

La famille de fonctions universelles telle qu'elle est construite est très arbitraire : elle dépend non seulement du choix des machines à registres pour représenter le calcul, mais aussi du choix du codage qui n'a rien d'intrinsèque.

On verra plus tard que la propriété d'énumération et l'existence de fonctions s_n^m, appelée aussi propriété de paramétrisation, axiomatisent les familles de fonctions récursives partielles universelles universelles : toute famille de fonctions universelles $(\psi^n)_{n \geq 1}$ qui vérifient ces deux propriétés se déduit de $(\varphi^n)_{n \geq 1}$ par une permutation récursive h :

$$\psi_i^1 = \varphi_{h(i)}^1 \ .$$

L'importance de la propriété de paramétrisation va apparaître dans les chapitres qui suivent. Avec les fonctions universelles, les entiers jouent un double rôle : celui de donnée, mais aussi celui de programme. La fonction s_n^m permet donc également de décrire des opérations calculables sur les programmes et de composer ceux-ci. Voyons un exemple simple. Soit e le code de la fonction $\lambda i\, j\, x.[\varphi_i^1(x) + \varphi_j^1(x)]$:

$$\varphi(i, x) + \varphi(j, x) = \varphi^3(e, i, j, x) = \varphi^3(s_1^2(e, i, j), x) \ .$$

La fonction $\lambda i\, j.s_1^2(e, i, j)$ calcule donc, à partir des indices de deux fonctions, l'indice de la somme de ces deux fonctions. Il fabrique en quelquesorte le (code du) programme pour la somme de ces deux fonctions, en intégrant comme sous-programmes les programmes (de codes i et j) calculant chacune de ces deux fonctions.

Table des matières