Équivalence MT-RAM

Damien NOGUÈS

13 février 2007

Table des matières

1 Introduction

Afin d'étudier la calculabilité, on a créé différents modèles de calcul, mais on s'est rendu compte que tous étaient équivalents aux machines de Turing (MT). C'est ce qui a donné la thèse de Church. Nous allons voir ici le modèle des RAM (Random Access Machines). Après avoir défini des RAM, nous montrerons l'équivalence avec les MT puis nous donnerons quelques éléments de complexité.

2 RAM : définition et fonctionnement

Une RAM (Random Access Machine) est constituée de différents registres, de deux bandes et d'un programme. Les deux bandes sont en fait une bande d'entrée sur laquelle la machine lit ses données et une bande de sortie sur laquelle la machine écrit ses résultats. Le programme, quant à lui, est une suite (finie) d'instructions.

2.1 Les registres

Les registres sont en quelque sorte des "cases mémoire" qui contiennent des entiers positifs. On distingue tout d'abord deux registres particuliers : le *compteur ordinal* qui contient le numéro de la prochaine instruction du programme à exécuter, ainsi qu'un registre spécial appelé *accumulateur*.

On dispose également d'une infinité de registres, indicés par $\mathbb{N}$. Pour représenter le n-ième registre, on notera : r_n.

2.2 Les instructions

Les instructions qui composent le programme se divisent elles aussi en quatre catégories :

2.2.1 Les instructions de manipulation de registres

Afin de manipuler les registres, on dispose de deux instructions `load` et `store`.
L'instruction `load` permet de stocker une valeur dans l'accumulateur. On peut s'en servir de trois façons différentes : (si on prend n un entier)

- `load #n` : (adressage absolu) cela a pour effet de mettre l'entier n dans l'accumulateur.
- `load n` : (adressage direct) cela a pour effet de stocker le contenu de r_n dans l'accumulateur.

- `load (n)` : (adressage indirect) si p est le contenu de r_n, alors cela a pour effet de stocker le contenu de r_p dans l'accumulateur.

L'instruction `store`, quant à elle, stocke le contenu de l'accumulateur dans un registre. Là aussi, il y a différents adressages possibles :

- `store n` : (adressage direct) cela enregistre le contenu de l'accumulateur dans r_n.

- `store (n)` : (adressage indirect) si p est le contenu de r_n, alors cela enregistre le contenu de l'accumulateur dans r_p.

2.2.2 Les opérations arithmétiques

On dispose aussi de deux opérations arithmétiques qui agissent sur l'accumulateur : `incr` et `decr`. L'opérateur `incr` incrémente l'accumulateur tandis que `decr` le décrémente s'il est positif et ne le modifie pas s'il est nul.

2.2.3 Les instructions de rupture de séquence

Cependant, une lecture linéaire du programme ne permettant pas de calculer de nombreuses fonctions, il nous faut des instructions dites de rupture de séquence. Tout d'abord, l'instruction `stop` qui provoque l'arrêt de la machine, ainsi que deux instructions de saut : un saut inconditionnel `jump` et un saut conditionnel `jumpz`.

Ces deux instructions prennent un entier en argument. Cet entier représente le numéro de la ligne où le programme doit "sauter" (cela revient à l'inscrire dans le compteur ordinal). La différence entre ces deux opérateurs est que `jumpz` ne provoque le saut que si l'accumulateur contient 0.

2.2.4 Les instructions d'entrée-sortie

Enfin, il faut pouvoir lire la bande d'entrée, ce qui se fait par l'instruction `read` qui lit un élément de la bande d'entrée et le stocke dans l'accumulateur puis déplace la tête de lecture vers l'entier suivant. Il faut aussi pouvoir écrire sur la bande de sortie le contenu de l'accumulateur, ce qui se fait avec l'instruction `write`.

2.3 Fonctionnement et exemple

Le fonctionnement de la machine est simple : on exécute successivement les instructions du programme (sauf évidemment pour les instructions qui suivent une instruction de saut) et on arrête le programme lorsque l'on rencontre l'instruction `stop` ou que l'on arrive à la fin du programme.

Exemple : programme qui calcule la somme de deux entiers :

0	`read`	lecture de x
1	`store 0`	x dans r_0
2	`read`	lecture de y
3	`store 1`	y dans r_1
4	`load 0`	boucle de calcul
5	`jumpz 12`	
6	`decr`	on décrémente x
7	`store 0`	
8	`load 1`	
9	`incr`	on incrémente y
10	`store 1`	
11	`jump 4`	fin de la boucle
12	`load 1`	
13	`write`	
14	`stop`	

3 Équivalence de la notion de calculabilité entre les MT et les RAM

Nous allons maintenant voir que, du point de vue de la calculabilité, les RAM sont équivalentes aux machines de Turing. Pour cela, la démonstration sera effectuée en deux étapes : tout d'abord, on tentera de simuler le fonctionnement d'une RAM sur une machine de Turing, puis l'inverse.

3.1 Simulation d'une machine de Turing par une RAM

La machine de Turing que nous allons simuler est une MT à une seule bande, déterministe, sous forme normalisée.

Pour simuler une machine de Turing à l'aide d'une RAM, la première chose à faire est de coder l'alphabet utilisé par la machine de Turing par des entiers. On supposera que ce codage concerne des entiers consécutifs de 0 à n et l'on supposera de plus que 0 codera le symbole # (symbole blanc de la bande).

Puis l'on conviendra que r_0 contiendra la position de la tête de lecture et que les registres de 1 à $+\infty$ stockeront la bande de la machine.

Tout d'abord, il faut initialiser nos registres, ce qui peut se faire avec le code suivant (on supposera que la bande d'entrée de notre RAM contient la bande initiale de notre MT) :

0	`load #1`	positionnement de la tête de lecture
1	`store 0`	sur le premier symbole de bande
2	`read`	
3	`jumpz 9`	arrêt si l'on rencontre un #
4	`store (0)`	sinon on stocke le symbole à l'endroit pointé par la tête de lecture
5	`load 0`	
6	`incr`	déplacement de la tête de lecture
7	`store 0`	
8	`jump 2`	
9	`load #1`	initialisation de la tête de lecture pour la simulation
10	`store 0`	

Puis, pour chaque état de la MT, il va falloir créer différents morceaux de code : tout d'abord, il nous faut charger le symbole lu avec la commande `load (0)`, puis à l'aide de décrémentations successives on va pouvoir déterminer quelle lettre a été lue et ainsi sauter à l'aide de sauts conditionnels vers la configuration adaptée.

En effet, on va ensuite créer un morceau de code pour chacune des lettres de notre alphabet qui mettra à jour la "bande", puis qui incrémentera ou décrémentera r_0 selon que l'on doive déplacer la tête de lecture à droite ou à gauche. Et qui enfin effectura le saut vers le numéro de ligne de l'état suivant dans le calcul.

Enfin, si l'on arrive dans notre état final (q^+), on écrit sur la bande de sortie l'état de notre "bande" simulée, et si on arrive dans une configuration impossible ou dans q^-, on écrit sur la bande de sortie un entier ne correspondant pas à un code de lettre pour montrer la non-reconnaissance du mot.

Si l'on fait de même avec tous les états de notre MT, on obtient bien une RAM qui reconnaît le même langage.

3.2 Simulation d'une RAM par une machine de Turing

Réciproquement, il est possible de simuler une RAM avec une machine de Turing.

Pour simplifier, on simulera la RAM sur une machine de Turing à trois bandes : sur la première bande, on écrira en binaire la valeur de l'accumulateur, suivie de toutes les valeurs des registres (jusqu'au registre de rang maximal qui aura été initialisé), le tout séparé par un caractère spécial, par exemple ','. Ainsi, à tout moment de l'exécution, la bande ne contiendra qu'un nombre fini de registres. La seconde bande, quant à elle, servira de compteur afin de pouvoir accéder au registre. Pour cela, il faut remonter la première bande jusqu'a son début, puis avancer le long de cette bande en

décrémentant à chaque fois que l'on recontre ','. Une fois arrivé à 0, il suffit de prendre le registre suivant (pour tenir compte de l'accumulateur en première position). Dans le cas l'où on arrive en bout de bande, on rajoute des 0 jusqu'à aller assez loin dans nos registres.

La troisième bande, quant à elle, ne servira qu'à stocker les valeurs à copier lors de l'appel des fonctions `load` et `store`.

Puis l'on associe à chaque instruction de notre programme une partie de MT qui correspond à la commande. Les opérations arithmétiques ne sont pas difficiles, elle reviennent à ajouter ou retrancher 1 au premier nombre écrit sur la bande. Les instructions de saut ne sont pas simulées mais elles déterminent la façon dont sont "branchées" les différentes parties de la MT (car il est assez aisé de verifier si l'accumulateur est nul).

4 À propos de complexité

Pour étudier la complexité sur les RAM, un problème se pose : en effet, une première approche pourrait être de considérer le nombre de registres utilisés pour la complexité en mémoire et le nombre d'opérations exécutées pour la complexité en temps. On appellera cette approche *uniforme*.

Cependant, les registres peuvent contenir de très grands entiers (la taille des entiers n'est pas bornée) ; dans ce cas, peut-on considérer que les opérations arithmétiques se font en temps constant ? Peut-on considérer qu'un registre qui contient 4 prend la même place qu'un registre qui contient $10^{10^{10}}$? On peut alors adopter une autre approche dite *logarithmique*.

4.1 La mesure de la complexité en temps

Il n'y a pas d'ambiguïté sur la mesure uniforme ; définissons alors la mesure logarithmique. Pour cela, on définit la taille d'un entier par :

$$t(n) = \left| \begin{array}{l} 1 \text{ si } n \leq 1 \\ \lceil log_2(n) \rceil + 1 \text{ sinon} \end{array} \right.$$

On conviendra alors que le temps mis par une opération qui manipule l'entier n est le produit des tailles des entiers manipulés lors de l'exécution de la fonction. Par exemple, lors de l'appel de `load (n)`, on obtient $t(n) \times t(r_n) \times t(x)$ où x est le contenu de l'accumulateur lors de l'appel. Ceci est le cas le "pire" : aucune instruction n'utilise plus d'entiers.

4.2 La mesure de la complexité en mémoire

De la même facon, on peut donner une mesure logarithmique de la complexité en mémoire. Définissons pour cela la fonction *size* : pour le registre i :

$$size(i) = \left| \begin{array}{l} 0 \text{ si } r_i \text{ n'est pas utilisé} \\ t(x) \text{ où x est la valeur contenue dans } r_i \text{ sinon} \end{array} \right.$$

On peut bien sûr prendre d'autres mesures (d'ailleurs, d'autres ont été proposées), mais avec cette mesure on peut montrer que l'on peut simuler notre RAM par une MT qui utilise la même complexité en mémoire (à un facteur multiplicatif près). Le problème est donc dans la complexité en temps.

4.3 Comparaison avec la complexité en temps d'une MT

Tout d'abord, on peut remarquer que la simulation d'une MT par un RAM se fait en multipliant le nombre d'étapes de calcul par un facteur qui peut être majoré (il ne dépend que de l'alphabet utilisé). On peut donc en déduire que, pour un problème donné, une RAM aura une classe de complexité (pour la mesure universelle) meilleure qu'une MT. Il a de plus été montré que, pour la mesure logarithmique, un facteur polynomial peut être ajouté.

De même, il existe un facteur polynomial qui, une fois multiplié à la complexité pour la machine de turing, devient supérieur à la complexité pour la machine RAM.

Donc, pour les classes de complexité au moins polynomiales les deux modèles sont équivalents.

5 Conclusion

Les RAM qui allient la puissance de calcul des MT avec un fonctionnement assez proche des ordinateurs réels peuvent donc fournir un outil intéressant pour l'étude de la calculabilité. Cependant, pour une étude fine de la complexité, ce modèle n'est pas forcement très adapté car il n'existe pas (à ma connaissance) de résultat qui permettent de comparer vraiment les classes de complexité.

6 Remerciements

Merci à Marc Sage pour la correction orthographique.