

M2 LFMI  
MEMOIRE DE STAGE

Unification des modèles de calculs caractérisant le temps polynomial

BEURIER Erwan

Année 2015-2016

# Table des matières

<b>1</b>	<b>Unification de deux caractérisations de <math>P</math></b>	<b>2</b>
1.1	Des caractérisations de $P$	2
1.1.1	Notions générales	2
1.1.2	Approche de Bellantoni et Cook	2
1.1.3	Approche de Cobham	2
1.1.4	Approche de Leivant	3
1.1.5	Correspondance entre Bellantoni et Cook et Leivant	4
1.2	Une histoire de RAMs	6
1.2.1	La $\sigma$ -RAM de Grandjean-Schwentick	6
1.2.2	La $\mathbb{A}$ -RAM de Leivant	6
1.2.3	Comparaison entre les deux RAM	7
1.3	Vers une caractérisation du temps polynomial sur $\sigma$ -RAM	13
1.3.1	Caractérisation du temps linéaire	13
	<b>Appendices</b>	<b>14</b>
.1	Simulation de la $\sigma$ -RAM par la $\mathbb{A}$ -RAM	15
.1.1	Conventions d'écriture.	15
.1.2	Ecriture des programmes	16

# Chapitre 1

## Unification de deux caractérisations de $P$

### 1.1 Des caractérisations de $P$

#### 1.1.1 Notions générales

- Algèbre
- Fonctions récursives sur une algèbre

#### 1.1.2 Approche de Bellantoni et Cook

L'idée de Bellantoni et Cook [1] est qu'on peut séparer les arguments d'une fonction en deux types. Le premier type, les arguments *normaux*, sont des arguments qu'on suppose bornés de manière implicite. Typiquement, un argument de récurrence est un argument normal : il ne faut pas, dans la définition par récurrence, que cet argument grossisse trop vite, car il risquerait d'entraîner un nombre croissant d'itérations. **Exemple de l'exponentielle (prendre à Leivant)**.

On se place dans une algèbre de mots (tous les constructeurs de l'algèbre sont d'arité au plus 1).

**Définition 1** (Fonctions récursives à arguments normaux [1]). *On note  $BC$  le plus petit ensemble de fonctions contenant :*

- les constructeurs *safe* :  $C_i (; x)$
- les projections :  $p_i^{k,h} (x_1, \dots, x_k; x_{k+1}, \dots, x_{k+h}) = x_i$
- le destructeur *safe* :  $dest (; C_i (; x)) = \begin{cases} C_i & \text{si } r_i = 0 \\ x & \text{sinon} \end{cases}$
- la conditionnelle *safe* :  $if (; x, y, z) = \begin{cases} y & \text{si } x \bmod 2 = 0 \\ z & \text{sinon} \end{cases}$

*et étant close par schéma de récursion safe*

$f$  est définie par récurrence *safe* à partir de  $(g_c)_{c \in \sigma} \in BC$  lorsque :

- $f (c(x), \bar{y}; \bar{z}) = g_c (x, \bar{y}; \bar{z}, f(x, \bar{y}; \bar{z}))$

*et composition safe.*

$f$  est définie par composition *safe* à partir de  $h, g_0, g_1 \in BC$  lorsque :

- $f (\bar{x}; \bar{y}) = h (\bar{x}, g_0(\bar{x}); \bar{y}, g_1(\bar{x}; \bar{y}))$

THÉORÈME 2 (Bellantoni et Cook). [1]  $BC = P$

#### 1.1.3 Approche de Cobham

Toujours dans le cas des mots :  $\sigma = \{C_1, \dots, C_k\}$  où  $\min_{i \in k} (C_i) = 0$  et  $\max_{i \in k} (C_i) = 1$ .

**Définition 3.** *On note  $Cob$  le plus petit ensemble de fonctions contenant les constructeurs, les projections, le smash  $x \# y = 2^{|x| \times |y|}$ , et close par composition et récurrence bornée sur les notations.*

Une fonction  $f \in Cob$  est définie par récurrence bornée sur les notations à partir de  $(g_c)_{c \in \sigma}, h \in Cob$  lorsque :

- $\forall c \in \sigma, \quad f (c(x), \bar{y}) = g_c (x, \bar{y})$
- $\forall x, \bar{y}, \quad |f (x, \bar{y})| \leq |h (x, \bar{y})|$

THÉORÈME 4 (Cobham).  $Cob = P$

Conséquence : il existe une borne explicite pour chaque fonction de  $P$ .

### 1.1.4 Approche de Leivant

Explication du point de départ de Leivant.

Résumé de l'article [3] de Leivant.

**Définition 5** (Algèbre [3]). Soit  $\sigma = \{C_1, \dots, C_k\}$  un ensemble de symboles de fonctions. Dans la suite,  $k$  représentera le nombre de constructeurs dans la signature.

La  $\sigma$ -algèbre  $\mathbb{A}$  est l'ensemble des termes clos constitués uniquement par les symboles de fonctions de  $\sigma$ . Dans ce cas, ces fonctions sont nommées constructeurs.

On note  $r_i$  l'arité du constructeur  $C_i$ . On suppose que  $\min_{r_i} (i \in [1, k]) = 0$  (sinon  $\mathbb{A}$  est vide) et  $\max_{r_i} (i \in [1, k]) = r > 0$  (sinon  $\mathbb{A}$  est finie).

Si  $r = 1$  alors  $\mathbb{A}$  est une algèbre de mots, sinon c'est une algèbre arborescente.

Soit un terme  $\tau \in \mathbb{A}$ . On appelle constructeur extérieur le dernier constructeur de  $\tau$ . On peut le définir par induction comme suit :

- Si  $\tau = C_i$  où  $r_i = 0$  alors  $C_i$  est le constructeur extérieur de  $\tau$ .
- Si  $\tau = C_i(\tau_1, \dots, \tau_{r_i})$  alors  $C_i$  est le constructeur extérieur de  $\tau$ .

**Exemples.**

- la  $\{0, s\}$ -algèbre des entiers unaires  $\mathbb{N}$ ;
- la  $\{\varepsilon, 0(-), 1(-)\}$ -algèbre des mots binaires  $\mathbb{W}$ .

On note  $(\mathbb{A}_i)_{i \in \omega}$  l'ensemble des copies de l'algèbre  $\mathbb{A}$ , qui correspondent à des niveaux d'abstraction de  $\mathbb{A}$ . On notera  $\mathcal{A}$  n'importe quel produit cartésien fini de copies de  $\mathbb{A}$ . Pour  $x \in \mathbb{A}$ , on note  $\text{tier}(x) = i$ . De même, pour  $f : \mathcal{A} \rightarrow \mathbb{A}_i$ , on note  $\text{tier}(f) = i$ . Chaque tier a de plus ses propres constructeurs. Le constructeur  $C_j$  du niveau  $i$  sera noté  $C_j^i$ .

**Définition 6.** On note  $T\text{Rec}(\mathbb{A})$  le plus petit ensemble de fonctions récursives primitives sur  $\mathbb{A}$  contenant les constructeurs, les projections et étant clos par composition ramifiée et récurrence ramifiée.

Une fonction est définie par récurrence ramifiée lorsque :

- $\forall i \quad f(C_i^j(a_1, \dots, a_{r_i}), \bar{x}) = g_{c_i}(f(a_1, \bar{x}), \dots, f(a_{r_i}, \bar{x}), \bar{a}, \bar{x})$

On note  $T\text{Rec}_2(\mathbb{A})$  le sous-ensemble de  $T\text{Rec}(\mathbb{A})$  dont chaque fonction est constructible en n'utilisant que deux tiers  $\mathbb{A}_0, \mathbb{A}_1$ .

Un résultat intéressant est que si une fonction  $f \in T\text{Rec}(\mathbb{A})$  a des arguments de tiers inférieur à son tier d'arrivée, alors  $f$  ignore ces arguments.

En plus de ça, on va définir une notion de degré d'imbrication, qui sert en somme de compteur de tours de boucles.

**Définition 7** (Degré d'imbrication de récurrences). Soit  $f \in T\text{Rec}(\mathbb{A})$ <sup>1</sup>.

Le degré d'imbrication de récurrence de  $f$ , noté  $\delta(f)$ , est un entier défini par induction sur la définition de  $f$  :

- Si  $f$  est un constructeur ou une projection, alors  $\delta(f) = 0$ .
- Si  $f$  est définie par composition, sans perte de généralité,  $f(\bar{x}) = g(\bar{x}, h(\bar{x}))$ , alors :
  - Si  $\text{tier}(h) < \text{tier}(g)$  alors  $\delta(f) = \delta(g)$  ;
  - Si  $\text{tier}(h) = \text{tier}(g)$  alors  $\delta(f) = \max(\delta(g), \delta(h))$  ;
  - Si  $\text{tier}(h) > \text{tier}(g)$  alors  $\delta(f) = \max(1, \delta(h)) \times \delta(g)$  ;
- Si  $f$  est définie par récurrence ramifiée  $f(C_i^j(a_1, \dots, a_{r_i}), \bar{x}) = g_{c_i}(f(a_1, \bar{x}), \dots, f(a_{r_i}, \bar{x}), \bar{a}, \bar{x})$ , telles que  $(g_{\alpha_j})_{j \in p}$  aient des arguments critiques et  $(g_{\beta_j})_{j \in q}$  n'en aient pas, alors  $\delta(f) = \max(1 + \delta(g_{\alpha_1}), \dots, 1 + \delta(g_{\alpha_p}), \delta(g_{\beta_1}), \dots, \delta(g_{\beta_q}))$ .

Il définit aussi une RAM personnalisée pour la construction de termes, voir 13. Cette RAM, notée  $\mathbb{A}$ -RAM dans ce mémoire, coïncide avec les autres modèles de calcul sur la classe  $P$ , mais son manque de test d'égalité et sa mémoire limitée la rendent moins puissante qu'une RAM au sens de Granjean [2] sur des classes plus fines (voir 1.2.3 pour la comparaison).

Les principaux résultats de Leivant ne concernent que les algèbres de mots. Voici les résultats de cet article [3] qui nous intéressent dans le cadre de ce mémoire :

1. Pour être rigoureux, le degré de récurrence ne s'applique pas à une fonction, mais à une *définition de fonction*.

THÉORÈME 8. Soit  $\mathbb{A}$  une algèbre de mots. Soit  $f : \mathcal{A} \rightarrow \mathbb{A}$ .

Les propositions suivantes sont équivalentes :

- $f$  est calculable en temps  $\mathcal{O}(n^k)$  sur une  $\mathbb{A}$ -RAM;
- $f$  est définissable par récurrence ramifiée sur deux tiers  $\mathbb{A}_0, \mathbb{A}_1$  et  $\delta(f) \leq k$ ;
- $f$  est définissable par récurrence ramifiée et  $\delta(f) \leq k$ .

THÉORÈME 9.  $TRec(\mathbb{A}) = TRec_2(\mathbb{A}) = P$

La récurrence ramifiée engendre  $P$  et deux niveaux de ramification suffisent.

Le théorème 8 a été prouvé pour une  $\mathbb{A}$ -RAM; l'un des objets de ce stage a aussi été d'obtenir un résultat similaire pour la  $\sigma$ -RAM.

### 1.1.5 Correspondance entre Bellantoni et Cook et Leivant

Le point de départ de Leivant est différent de celui de Bellantoni et Cook, mais le résultat est similaire. En fait, il y a une correspondance naturelle entre le résultat de Bellantoni et Cook et celui de Leivant.

LEMME 10. Soit  $f \in P$ ,  $f$  fonction sur  $\mathbb{W}$  (mots binaires).

Un argument  $x$  de  $f$  est normal dans  $BC$  si et seulement si  $\text{tier}(x) > \text{tier}(f)$  dans  $TRec(\mathbb{W})$ .

Un argument  $x$  de  $f$  est safe dans  $BC$  si et seulement si  $\text{tier}(x) \leq \text{tier}(f)$  dans  $TRec(\mathbb{W})$ .

Démonstration.

**Passage de BC à TRec( $\mathbb{W}$ )**

Un constructeur *safe*  $C_i(;x)$  se traduit naturellement en un constructeur *plat* (le tier de départ est égal au tier d'arrivée).

Les projections se traduisent naturellement aussi en suivant cette règle.

Le destructeur *safe*  $\text{dest} (; C (; x))$  est définissable par récurrence plate (sans appel récursif) dans  $TRec(\mathbb{W})$  :

$$\text{dest}(C_i(x)) = \begin{cases} C_i & \text{si } r_i = 0 \\ x & \text{sinon} \end{cases}$$

Une ramification triviale donne  $\text{dest} : \mathbb{W}_i \rightarrow \mathbb{W}_i$ .

De même, la conditionnelle *safe*  $\text{if} (; x, y, z)$  est définissable par récurrence plate dans  $TRec(\mathbb{W})$  :

$$\begin{aligned} \text{case}(\varepsilon, y, z) &= y \\ \text{case}(0(x), y, z) &= y \\ \text{case}(1(x), y, z) &= z \end{aligned}$$

Une ramification triviale donne  $\text{case} : \mathbb{W}_i^3 \rightarrow \mathbb{W}_i$ .

Soit  $f$  définie par composition *safe*  $f(\bar{x}; \bar{y}) = h(\bar{x}, g_n(\bar{x}); \bar{y}, g_s(\bar{x}; \bar{y}))$ .

Par hypothèse d'induction,

$$\begin{aligned} g_n : \overline{\mathbb{W}_{k_n}} &\rightarrow \mathbb{W}_{i_n} && \text{avec } k_n > i_n \\ g_s : \overline{\mathbb{W}_{k_s}} \times \overline{\mathbb{W}_{j_s}} &\rightarrow \mathbb{W}_{j_s} && \text{avec } k_s > j_s \\ h : \overline{\mathbb{W}_{k_h}} \times \overline{\mathbb{W}_{i_h}} \times \overline{\mathbb{W}_{j_h}} \times \overline{\mathbb{W}_{j_h}} &\rightarrow \mathbb{W}_{j_h} && \text{avec } k_h, i_h > j_h \end{aligned}$$

Remarquons d'abord que, si une fonction  $f_{i,j} : \mathbb{W}_i \rightarrow \mathbb{W}_j$  est définissable par récurrence ramifiée en utilisant des tiers dans  $[0, k]$ , alors il existe  $f_{i+h, j+h} : \mathbb{W}_{i+h} \rightarrow \mathbb{W}_{j+h}$  définissable par récurrence ramifiée utilisant des tiers dans  $[h, k+h]$  pour tout  $h \in \omega$  (la démonstration se fait par une induction automatique sur la construction de  $f_{i,j}$ ). De plus, il existe  $f_{i+h, j} : \mathbb{W}_{i+h} \rightarrow \mathbb{W}_j$  pour tout  $h \in \omega$  (il suffit pour cela d'utiliser une fonction dite de coercion  $\kappa_{i,j} : \mathbb{W}_i \rightarrow \mathbb{W}_j, i > j$ , qui convertit un élément d'un tier en un tier inférieur).

De ce fait, on peut choisir  $g_n, g_s$  et  $h$  telles que les tiers correspondent, à savoir :  $i_n = i_h = i$ ,  $j_h = j_s = j$  et  $k_h = k_n = k_s = k$ . Le résultat découle naturellement :  $f : \overline{\mathbb{W}_k} \times \overline{\mathbb{W}_j} \rightarrow \mathbb{W}_j$ .

Pour la définition par récurrence, le raisonnement est similaire.

**Passage de TRec( $\mathbb{W}$ ) à BC**

LEMME 11 (Leivant [3]). Soit  $f : \mathbb{A}_i \times \mathcal{A} \rightarrow \mathbb{A}_j$ . Si  $i < j$  alors  $\forall x, f(x, \bar{x}) = f(C, \bar{x})$  où  $C$  est un constructeur d'arité 0 de  $\mathbb{A}$ .

Si l'argument d'une fonction est d'un tier inférieur à celui de la fonction, alors cet argument est inutile à la fonction. En un sens, il n'est pas assez abstrait. Un tel argument est dit *redondant*.

Dans la suite, on suppose que les fonctions utilisées n'ont pas d'arguments redondants. De plus, pour simplifier, on va supposer qu'on n'utilise que deux tiers  $\mathbb{A}_0, \mathbb{A}_1$  (ce qui est permis d'après le théorème 9).

Reprenons la démonstration du passage de Leivant à Bellantoni et Cook.

Le cas des constructeurs et des projections est trivial.

Si  $f(\bar{x}, \bar{y}) = g(\bar{x}, h_1(\bar{x}), \bar{y}, h_0(\bar{x}, \bar{y}))$  alors :

- Soit  $\text{tier}(f) = \text{tier}(g) = 1$ , dans ce cas  $f(\bar{x}) = g(\bar{x}, h_1(\bar{x}))$  (car on n'a pas d'argument redondant). Par hypothèse d'induction  $h_1(\bar{x}) = h_1(; \bar{x})$  et  $g(\bar{x}, h_1(\bar{x})) = g(; \bar{x}, h_1(; \bar{x})) = f(; \bar{x})$ .
- Soit  $\text{tier}(f) = \text{tier}(g) = 0$ , dans ce cas par hypothèse d'induction, on a bien :  $g(\bar{x}, h_1(\bar{x}); \bar{y}, h_0(\bar{x}, \bar{y})) = f(\bar{x}; \bar{y})$ .

Le cas de la récurrence ramifiée se résout de manière analogue. ■

□

L'équivalence entre les deux se généralise sans problème aux algèbres de mots voire aux algèbres quelconques (bien que dans le cas d'algèbres quelconques, on ne sache pas s'il y a équivalence entre  $P$  et  $BC$  ou  $TRec(\mathbb{A})$ ).

## 1.2 Une histoire de RAMs

### 1.2.1 La $\sigma$ -RAM de Grandjean-Schwentick

Soit  $\sigma$  une signature fonctionnelle unaire ou binaire, typiquement  $\sigma = \{+, -, \times, \div 2\}$ .

**Définition 12** ( $\sigma$ -RAM [2]). *Une  $\sigma$ -RAM est un modèle de calcul composé de :*

- Deux accumulateurs  $A, B$  ;
- Un registre spécial  $N$  ;
- Une infinité dénombrable de registres  $(R_i)_{i \in \omega}$ .

*Les registres contiennent a priori des valeurs entières.*

*Un programme de  $\sigma$ -RAM est un ensemble fini d'instructions  $(I(i))_{i \in N}$  dont chacune est de l'une des formes suivantes :*

- $A := c$  pour n'importe quelle constante  $c \in \mathbb{N}$
- $A := op(A)$  ou  $op(A, B)$ , où  $op \in \sigma$
- $A := N$
- $N := A$
- $A := R_A$
- $B := A$
- $R_A := B$
- $IF(A = B) \{I(i)\} \text{ ELSE } \{I(j)\}$
- $HALT$

**Explications.** Ces instructions sont assez claires.  $R_A$  est le registre  $R_j$  où  $j$  est la valeur contenue dans l'accumulateur  $A$ . La commande  $IF$  renvoie à l'instruction  $i$  si  $A = B$  et à  $j$  sinon. Par défaut, tous les registres sont initialisés à 0.

Cette  $\sigma$ -RAM est déterministe. On pourrait la rendre non-déterministe en autorisant une commande  $A := CHOOSE(A)$  ou une commande  $GOTO(I(i_1), \dots, I(i_t))$  qui permet d'effectuer plusieurs instructions en même temps.

**Entrées et sorties.** Dans leur article [2], Grandjean et Schwentick indiquent que leur  $\sigma$ -RAM prend en entrée une

structure  $\left( [1, n], \underbrace{f_1, \dots, f_k}_{\text{fonctions unaires}}, \underbrace{C_1, \dots, C_l}_{\text{constantes}} \right)$  et s'initialise de la façon suivante :

- $N := n$
- $R_{(i-1) \times n + j} := f_i(j)$  où  $i \in [1, k]$  et  $j \in [1, n]$
- $R_{k \times n + j} := C_i$  où  $i \in [1, l]$

(La taille est stockée dans  $N$  et les fonctions sont stockées dans l'ordre de numérotation).

La sortie est une nouvelle structure  $\left( [1, n'], \underbrace{f'_1, \dots, f'_{k'}}_{\text{fonctions unaires}}, \underbrace{C'_1, \dots, C'_{l'}}_{\text{constantes}} \right)$  qui remplace l'entrée.

### 1.2.2 La $\mathbb{A}$ -RAM de Leivant

Soit  $\mathbb{A}$  une  $\sigma$ -algèbre.

**Définition 13** ( $\mathbb{A}$ -RAM [3]). *Une  $\mathbb{A}$ -RAM est un modèle de calcul comprenant :*

- Un ensemble fini d'états  $S = \{s_1, \dots, s_l\}$ , où  $s_1$  est l'état initial et  $s_l$  est l'état final ;
- Un ensemble fini de registres  $\Pi = \{\pi_1, \dots, \pi_m\}$ .

*Les registres contiennent des termes de l'algèbre  $\mathbb{A}$ . Par défaut, on leur assigne une valeur  $d$  parmi les constructeurs d'arité 0.*

*Un programme de  $\mathbb{A}$ -RAM est un ensemble fini d'instructions dont chacune est de l'une des formes suivantes :*

- $(const) s_a \pi_{j_1} \dots \pi_{j_r} C_i \pi_j s_b$
- $(p-dest) s_a \pi_i \pi_j s_b$
- $(switch) s_a \pi_j s_{b_1} \dots s_{b_k}$

**Explications.** Pour des raisons de lisibilité et de simplification d'écriture, on notera  $*j$  le contenu du registre  $\pi_j$ .

La commande (const)  $s_a \pi_{j_1} \dots \pi_{j_{r_i}} C_i \pi_j s_b$  peut se lire : si la  $\mathbb{A}$ -RAM est dans l'état  $s_a$  alors  $\pi_j := C_i(*j_1, \dots, *j_{r_i})$  (on construit un nouveau terme  $C_i(*j_1, \dots, *j_{r_i})$  que l'on place dans  $\pi_j$ ). Après avoir effectué l'instruction, la  $\mathbb{A}$ -RAM passe à l'état  $s_b$ .

La commande (p-dest)  $s_a \pi_i \pi_j s_b$  permet, si la  $\mathbb{A}$ -RAM est dans l'état  $s_a$ , de récupérer le  $p$ -ième argument du constructeur extérieur du terme  $*j$  pour le stocker dans  $\pi_j$ . Si on détruit un constructeur d'arité 0, alors on récupère la valeur par défaut. Après avoir effectué l'instruction, la  $\mathbb{A}$ -RAM passe à l'état  $s_b$ . Notons que le label (p-dest) est indispensable pour savoir quel sous-terme récupérer.

La commande (switch)  $s_a \pi_j s_{b_1} \dots s_{b_k}$  lit  $\pi_j$  et, selon le résultat, place la  $\mathbb{A}$ -RAM dans l'état  $s_{b_i}$  si le constructeur extérieur de  $*j$  est  $C_i$ .

Cette  $\mathbb{A}$ -RAM est déterministe lorsque, à chaque état  $s_a$ , correspond une unique commande.

**Entrées et sorties.** Les entrées et les sorties sont des termes de  $\mathbb{A}$ . Si la  $\mathbb{A}$ -RAM prend  $k$  entrées, alors ces entrées sont stockées dans les  $k$  premiers registres de la machine. La sortie se trouve dans le dernier registre  $\pi_m$ .

### 1.2.3 Comparaison entre les deux RAM

#### Modèles Turing-complets

**THÉORÈME 14.** Une machine de Turing (binaire ?) simule une  $\sigma$ -RAM (pour quel  $\sigma$  ?) en temps ??? et une  $\sigma$ -RAM simule une machine de Turing en temps polynomial.

Une  $\mathbb{W}$ -RAM simule une machine de Turing binaire en temps linéaire et une machine de Turing binaire simule une  $\mathbb{W}$ -RAM en temps polynomial.

Les démonstrations se trouvent dans [3] Il me faut une source pour la  $\sigma$ -RAM.

#### $\mathbb{A}$ -RAM dans $\sigma$ -RAM

Le but de cette section est de simuler une  $\mathbb{A}$ -RAM dans une  $\sigma$ -RAM et inversement.

**Premières observations.** Premièrement, la  $\sigma$ -RAM calcule des entiers alors que la  $\mathbb{A}$ -RAM construit des termes. Cette remarque plutôt intuitive est pourtant la cause *meta* des différences de puissance de ces machines. La  $\sigma$ -RAM se veut un modèle de calcul très proche de l'ordinateur réel (la mémoire réelle n'est certes pas infinie, mais suffisamment grande pour que ça ne soit pas un problème), alors que la  $\mathbb{A}$ -RAM se veut un modèle plus logique (construction de termes). De plus, la machine de Leivant est très frustrée car il l'a construite pour qu'elle colle parfaitement à sa caractérisation de  $DTIME_{\mathbb{A}}(n^k)$ .

Deuxièmement, la mémoire de la  $\sigma$ -RAM est infinie et celle de la  $\mathbb{A}$ -RAM est finie et ne doit pas dépendre de l'entrée. Troisièmement, la  $\sigma$ -RAM dispose d'un test d'égalité gratuit, ce qui n'est pas le cas de la  $\mathbb{A}$ -RAM. Enfin, la  $\mathbb{A}$ -RAM a une commande de destruction de terme qui n'a pas réellement d'équivalent dans la  $\sigma$ -RAM.

Afin de contourner le premier problème, on va étendre la définition de la  $\sigma$ -RAM pour qu'elle construise elle aussi des termes. Quant aux autres problèmes, il va falloir ruser. Pour la mémoire, on va utiliser une astuce et distinguer quelques registres qui contiendront des termes très grands - mais qui ralentiront l'accès à la mémoire. Le test d'égalité sera simulé lourdement par la  $\mathbb{A}$ -RAM et on rajoutera une commande en plus à la  $\sigma$ -RAM pour copier la destruction de terme.

Soit  $\sigma = \{C_1, \dots, C_k\}$  une signature de constructeurs. On note  $r = \max_{r_i} (i \in [1, k])$ . On suppose que  $r > 0$ .

Notons  $\mathbb{A}$  la  $\sigma$ -algèbre contenant ces termes.

**Définition 15** ( $\sigma$ -RAM constructrice). Une  $\sigma$ -RAM constructrice est un modèle de calcul composé de :

- $r$  accumulateurs  $(A_i)_{i \in [1, r]}$  (si  $r = 1$ , on suppose qu'on a au moins deux accumulateurs) ;
- Un registre spécial  $N$  ;
- Une infinité dénombrable de registres  $(R_i)_{i \in \omega}$ .

Les instructions sont similaires à celles de la  $\sigma$ -RAM classique :

- $A_1 := c$  pour n'importe quelle constante  $c \in \mathbb{A}$
- $A_1 := C_i(A_1, \dots, A_{r_i})$  où  $C_i \in \sigma$
- $A_1 := N$
- $A_1 := \text{dest}_p(A_1)$  où  $p \in [1, r]$
- $N := A_1$



- $A_1 := R_{A_1}$
- $A_i := A_1$  où  $i \in [2, \max(2, r)]$
- $R_{A_1} := A_i$  où  $i \in [2, \max(2, r)]$
- $IF(A_1 = A_2)\{I(i)\} \text{ ELSE } \{I(j)\}$
- $HALT$

On garde les commandes classiques et on rajoute une commande de destruction  $A_1 := \text{dest}_p(A_1)$  qui a exactement le même principe que la commande analogue dans la  $\mathbb{A}$ -RAM.

Pour donner du sens à  $R_{A_1}$ , deux choix s'offrent à nous. Soit on suppose que les registres sont numérotés par des entiers naturels et, puisque  $\mathbb{A}$  est dénombrable, on a une bijection entre les deux. On suppose alors que  $R_{A_1}$  est le registre dont l'indice est l'image du terme contenu dans  $A_1$  par la bijection. Sinon, on peut considérer que les registres sont numérotés directement par des termes.

Dans la suite, on va se placer dans le cas où l'on a une bijection  $f : \mathbb{N} \rightarrow \mathbb{A}$ . De plus, par abus de langage, on parlera de  $\sigma$ -RAM sans préciser si elle est constructrice ou non (elle le sera par défaut).

**Simulation.** On considère la signature  $\sigma = \{C_1, \dots, C_k\}$  et la  $\sigma$ -algèbre  $\mathbb{A}$ .

LEMME 16. Une  $\mathbb{A}$ -RAM peut être simulée en temps linéaire par une  $\sigma$ -RAM.

*Démonstration.* Les registres de la  $\mathbb{A}$ -RAM seront simplement simulés par les registres de la  $\sigma$ -RAM.

Si la  $\mathbb{A}$ -RAM est déterministe, alors ses états n'ont qu'un seul état suivant. Dans ce cas, le passage d'un état à un autre est simulé dans la  $\sigma$ -RAM comme simplement le passage d'un ensemble d'instructions à un autre. Si la  $\mathbb{A}$ -RAM est non-déterministe, alors on peut passer à plusieurs états en même temps. Pour simuler cela, on autorise une commande  $GOTO(I(j_1), \dots, I(j_t))$  qui permet d'aller aux instructions correspondantes.

Il suffit de traduire les instructions de la  $\mathbb{A}$ -RAM en suite d'instructions de la  $\sigma$ -RAM.

**(const)**  $s_a \pi_{j_1} \dots \pi_{j_{r_i}} C_i \pi_j s_b$

```
// Dans la boucle, on récupère les arguments voulus :
for t from  $r_i$  to 1 do
   $A_1 := f(j_t)$ ;
   $A_1 := R_{A_1}$ ;
   $A_t := A_1$ ;
end
// On construit le terme :
 $A_1 := C_i(A_1, \dots, A_{r_i})$ ;
// On stocke le résultat dans  $R_{f(l)}$  :
 $A_2 := A_1$ ;
 $A_1 := f(l)$ ;
 $R_{A_1} := A_2$ ;
```

**Algorithm 1:** Simulation de la commande (const)

Cette simulation se fait en  $3r_i + 4$  étapes de calcul.

**(p-dest)**  $s_a \pi_i \pi_j s_b$

```
// On récupère le contenu de  $R_{f(i)}$  :
 $A_1 := f(i)$ ;
 $A_1 := R_{A_1}$ ;
// On récupère son  $p$ -ième sous-terme (on effectue l'opération de destruction) :
 $A_1 := \text{dest}_p(A_1)$ ;
// On déplace le résultat dans  $R_{f(j)}$  :
 $A_2 := A_1$ ;
 $A_1 := f(j)$ ;
 $R_{A_1} := A_2$ ;
```

**Algorithm 2:** Simulation de la commande (p-dest)

Cette simulation se fait en 6 étapes de calcul.

(switch)  $s_a \pi_j s_{b_1} \dots s_{b_k}$

```

// On détruit entièrement le terme et on récupère chacun des arguments de son constructeur
// extérieur de  $R_{f(j)}$ . Si l'arité du constructeur est strictement inférieure à  $p$ , on assigne une
// valeur par défaut au registre.
for  $p$  from  $r$  to 1 do
   $A_1 := f(j)$ ;
   $A_1 := R_{A_1}$ ;
   $A_p := \text{dest}_p(A_1)$ ;
end
for  $i$  from 1 to  $k$  do
  // On construit un terme :
   $A_1 := C_i(A_1, \dots, A_{r_i})$ ;
  // On le met de côté, à disposition, quitte à écraser le contenu de  $A_2$  :
   $A_2 := A_1$ ;
  // On récupère le terme qu'on voulait analyser :
   $A_1 := f(j)$ ;
   $A_1 := R_{A_1}$ ;
  if  $A_1 = A_2$  then
    // Alors on a reconstruit le même terme ; ça veut dire qu'on est au bon  $i$ . Dans ce cas, on
    // va à l'instruction  $j_i$  :
    IF( $A_1 = A_2$ ){ $I(j_i)$ } ELSE { $I(j_i)$ };
    // Astuce pour avoir un GOTO
  else
    // Sinon, on s'est trompé. Dans ce cas, on récupère l'argument qu'on venait d'effacer dans
    //  $A_2$  :
     $A_1 := \text{dest}_2(A_1)$ ;
     $A_2 := A_1$ ;
    // On récupère aussi le premier sous-terme, pour pouvoir faire une nouvelle construction
    // par la suite :
     $A_1 := f(j)$ ;
     $A_1 := R_{A_1}$ ;
    // Et on continue la boucle.
  end
end
end
// On construit un nouveau terme en regardant chaque constructeur et on vérifie s'il est égal au
// terme de  $R_{f(j)}$ 

```

**Algorithm 3:** Simulation de la commande (switch)

La simulation se fait en  $\leq 3r + 9k$  étapes. Encore une fois, les boucles *For* sont des simplifications d'écriture.

**Bilan.** Chaque instruction de la  $\mathbb{A}$ -RAM peut être simulée en temps constant par une  $\sigma$ -RAM. ■

□

### $\sigma$ -RAM dans $\mathbb{A}$ -RAM

LEMME 17. Une  $\sigma$ -RAM fonctionnant en temps polynomial peut être simulée en temps polynomial par une  $\mathbb{A}$ -RAM.

*Démonstration.* Ici, nous allons surtout présenter les algorithmes. La programmation en  $\mathbb{A}$ -RAM est disponible en annexes. Avant d'en venir à la simulation proprement dite, on va d'abord décrire quelques fonctions.

**Copie** La  $\mathbb{A}$ -RAM permet une copie indépendante de la longueur du terme.

**Input:**  $\alpha, \beta, \pi_1, \dots, \pi_r$

// Détruit le terme contenu dans  $\alpha$ , stocke ses composantes dans chaque  $\pi_p$ , puis reconstruit le terme dans  $\beta$ .

**for**  $p$  from 1 to  $r$  **do**

  | Récupérer le  $p$ -ième sous-terme de  $\alpha$  et le stocker dans  $\pi_p$  ;

**end**

Selon le constructeur extérieur de  $\alpha$ , utiliser le même constructeur pour construire dans  $\beta$  l'exact même terme, utilisant les mêmes sous-termes stockés dans les  $\pi_1, \dots, \pi_r$ .

**Algorithm 4:** La fonction  $s_{a_1}\text{COPY}(\alpha, \beta, \bar{\pi})s_b$ . Programme ici.

La copie se fait en temps constant  $(r + 2)$ .

**Egalité.** Pour gérer le premier problème, on va écrire un test d'égalité. Puisqu'on ne peut comparer que les constructeurs extérieurs des termes, on va effectuer ces comparaisons, puis déconstruire le terme et recommencer les comparaisons. Puisque le nombre de sous-termes peut dépasser la capacité (finie) d'une  $\mathbb{A}$ -RAM, on va utiliser un fragment de l'astuce ci-dessous concernant la mémoire : à chaque décomposition de terme, on stockera les sous-termes dans une mémoire en forme de liste.

**Input:**  $\alpha, \beta, s_1, s_0, \pi_1, \pi_2, \pi'_1, \pi'_2$

/\*  $\alpha, \beta$  : les registres contenant les termes à tester. \*/

/\*  $s_1, s_0$  : place la machine dans l'état  $s_1$  si  $\alpha = \beta$ ,  $s_0$  sinon. \*/

/\*  $\pi_1, \pi_2$  : registres de travail ; contiendront respectivement la liste des sous-termes de  $\alpha$  et  $\beta$  qui n'ont pas encore été comparés. \*/

/\*  $\pi'_1, \pi'_2$  : registres de travail ; contiendront les sous-termes courants. \*/

/\* \*/

/\* Vérifie si  $\alpha = \beta$  en faisant une analyse inductive sur la construction des termes qu'ils contiennent. \*/

Copier  $\alpha$  dans  $\pi'_1$  ;

Copier  $\beta$  dans  $\pi'_2$  ;

// C'est une boucle do...while à la C++ ; on doit passer au moins une fois dedans.

**while**  $\pi_1$  et  $\pi_2$  sont non vides **do**

**if**  $\pi'_1$  et  $\pi'_2$  ont les mêmes constructeurs extérieurs **then**

**if**  $\pi'_1$  et  $\pi'_2$  ont des sous-termes **then**

      Empiler les sous-termes de  $\pi'_1$  dans  $\pi_1$  ;

      Empiler les sous-termes de  $\pi'_2$  dans  $\pi_2$  ;

**end**

    Récupérer le premier élément de  $\pi_1$  et le stocker dans  $\pi'_1$  ;

    Récupérer le premier élément de  $\pi_2$  et le stocker dans  $\pi'_2$  ;

    Dépiler  $\pi_1$  ;

    Dépiler  $\pi_2$  ;

**else**

    Aller à l'état  $s_1$  ;

**end**

**end**

Aller à l'état  $s_0$  ;

**Algorithm 5:** Fonction  $\text{IF}(\alpha, \beta, s_1, s_0, \pi_1, \pi_2, \pi'_1, \pi'_2)$ . Programme ici.

Cette fonction fait le test d'égalité en temps  $\mathcal{O}(\min(|\alpha|, |\beta|))$ . Ce n'est donc pas un test gratuit.

**Mémoire.** Pour le second problème, on va supposer que la  $\mathbb{A}$ -RAM a le droit d'utiliser deux constructeurs supplémentaires  $\text{MEM}(-, -, -)$  et  $\varepsilon$ . La mémoire de la  $\sigma$ -RAM sera simulée par un terme de la forme :

$\text{MEM}(f(i_1), R_{f(i_1)}, \text{MEM}(\dots, \text{MEM}(f(i_n), R_{f(i_n)}, \varepsilon) \dots))$

Le premier argument de  $\text{MEM}$  est l'indice du registre, le deuxième est son contenu, et le troisième la suite de la mémoire. Notons qu'on ne suppose pas que les indices sont ordonnés, car ils ne pourront pas l'être.

Le but est de pouvoir reproduire la mémoire de la  $\sigma$ -RAM en autorisant un (ou plusieurs) terme(s) de longueur non bornée.

Pour utiliser cette mémoire, on va distinguer cinq registres :  $\mu$  et  $(\mu_i)_{i \in 4}$ .

$\mu_1$  contiendra l'indice courant de la mémoire,  $\mu_2$  contiendra la valeur du registre associé.  $\mu_3$  contiendra la mémoire suivante et  $\mu_4$  la mémoire précédente.  $\mu$  est copiée dans  $\mu_3$  à l'initialisation de certaines fonctions, puis  $\mu_3$  est déroulée, en stockant le contenu passé dans  $\mu_4$ .

On définit des fonctions associées qui permettront de mieux comprendre comment on se sert de cette mémoire.

```

Input:  $\mu, \mu_1, \mu_2, \mu_3, \mu_4, \alpha, \beta, \pi_1, \pi_2, \pi'_1, \pi'_2$ 
/*  $\mu, (\mu_i)_{i \in 4}$  sont tels que décrits plus haut. */
/*  $\alpha$  est l'indice auquel on veut insérer. */
/*  $\beta$  est la valeur qu'on veut insérer à l'indice  $\alpha$ . */
/*  $\pi_1, \pi_2, \pi'_1, \pi'_2$  sont les registres de travail du test d'égalité. */

// Initialisation de la mémoire
Copier  $\mu$  dans  $\mu_3$  ;
Vider  $\mu_4$  ;
Stocker le premier indice et la première valeur de  $\mu$  dans  $\mu_1$  et  $\mu_2$  ;

// Première boucle : on avance dans la mémoire en vérifiant à chaque fois si on est au bon
// indice.
while  $\mu_3$  est non-vide do
    Empiler  $(\mu_1, \mu_2)$  dans  $\mu_4$  ;
    Stocker le premier indice de  $\mu_3$  dans  $\mu_1$  ;
    Stocker la première valeur de  $\mu_3$  dans  $\mu_2$  ;
    Dépiler  $\mu_3$  ;
    if  $\alpha = \mu_1$  then
        | Sortir de la boucle While ;
    end
end
Empiler  $(\mu_1, \beta)$  dans  $\mu_3$  ;

// Deuxième boucle : étape inverse : on récupère tout ce qu'on a visité précédemment.
while  $\mu_4$  est non-vide do
    Empiler  $(\mu_1, \mu_2)$  dans  $\mu_3$  ;
    Stocker le premier indice de  $\mu_4$  dans  $\mu_1$  ;
    Stocker la première valeur de  $\mu_4$  dans  $\mu_2$  ;
    Dépiler  $\mu_4$  ;
end
Copier  $\mu_3$  dans  $\mu$  ;

```

**Algorithm 6:** Fonction INSERT  $(\mu, \mu_1, \mu_2, \mu_3, \mu_4, \alpha, \beta, \pi_1, \pi_2, \pi'_1, \pi'_2)$ . Programme ici.

La fonction INSERT fonctionne en temps  $\leq \mathcal{O}(|\mu| \times \max(|\mu|, |\alpha|))$ .

```

Input:  $\mu, \mu_1, \mu_2, \mu_3, \mu_4, \alpha, \beta, \pi_1, \pi_2, \pi'_1, \pi'_2$ 
/*  $\mu, (\mu_i)_{i \in 4}$  sont tels que décrits plus haut. */
/*  $\alpha$  est l'indice auquel on veut insérer. */
/*  $\beta$  est la valeur qu'on veut insérer à l'indice  $\alpha$ . */
/*  $\pi_1, \pi_2, \pi'_1, \pi'_2$  sont les registres de travail du test d'égalité. */
// Initialisation de la mémoire
Copier  $\mu$  dans  $\mu_3$  ;
Vider  $\mu_4$  ;
Stocker le premier indice et la première valeur de  $\mu$  dans  $\mu_1$  et  $\mu_2$  ;

// Boucle : avancer dans la mémoire jusqu'à trouver le bon indice.
while  $\mu_3$  est non-vide do
    Stocker le premier indice de  $\mu_3$  dans  $\mu_1$  ;
    Stocker la première valeur de  $\mu_3$  dans  $\mu_2$  ;
    Dépiler  $\mu_3$  ;
    if  $\alpha = \mu_1$  then
        | Sortir de la boucle While ;
    end
end
Copier  $\mu_2$  dans  $\beta$  ;

```

**Algorithm 7:** Fonction ACCESS ( $\mu, \mu_1, \mu_2, \mu_3, \mu_4, \alpha, \beta, \pi_1, \pi_2, \pi'_1, \pi'_2$ ). Programme ici.

La fonction ACCESS fonctionne en temps  $\leq \mathcal{O}(|\mu| \times \max(|\alpha|, |\text{longueur des adresses déjà employées}|))$ .

**Simulation.** Pour  $\mathbb{A}$  engendrée par  $k$  constructeurs  $\sigma = \{C_1, \dots, C_k\}$  d'arité maximale  $r$ , la  $\sigma$ -RAM est constituée de  $r$  accumulateurs  $A_1, \dots, A_r$  (si  $r = 1$ , on suppose qu'il y en a au moins 2), un registre spécial  $N$  et une infinité de registres  $(R_i)_{i \in \omega}$ . Pour simuler cette  $\sigma$ -RAM, on va utiliser une  $\mathbb{A}$ -RAM contenant un registre spécial pour chaque accumulateur  $(\alpha_i)_{i \in k}$ , un registre  $\nu$ , cinq registres spéciaux dédiés à la gestion de la mémoire  $\mu, (\mu_i)_{i \in 4}$ ,  $r + 4$  registres de travail  $\pi'', \pi_1, \pi_2, \pi'_1, \pi'_2$ .

Les états de la  $\mathbb{A}$ -RAM seront simplement calqués sur les numéros des instructions du programme de la  $\sigma$ -RAM. Si une instruction de  $\sigma$ -RAM se simule en un programme de  $\mathbb{A}$ -RAM, alors l'état initial de cette simulation est l'état correspondant au numéro de l'instruction, et l'état final correspond au numéro de l'instruction suivante. Les états intermédiaires dans les fonctions sont des états qui n'apparaissent nulle part ailleurs que dans la fonction associée, afin d'éviter les conflits.

**Entrées/sorties** Dans une  $\sigma$ -RAM, les entrées sont stockées dans des registres au choix. Il suffit de reproduire cette initialisation en remplissant les registres équivalents dans la  $\mathbb{A}$ -RAM. Si  $A_i, N$  sont initialisés, alors initialiser  $\alpha_i, \nu$  de la même façon. Si des registres  $R_i$  sont initialisés, alors on initialise  $\mu$  avec un terme mémoire :

MEM( $f(0), R_0, \text{MEM}(f(1), R_1, \dots)$ ).

Dans la suite,  $(a)$  sera le numéro de l'instruction étudiée.

**Assignation de terme.**  $(a)$   $A := c$  où  $c \in \mathbb{A}$  est un terme indépendant du calcul, se simule en temps constant en reconstruisant *manuellement* le terme  $c$  dans la  $\mathbb{A}$ -RAM.

**Instructions de copie.**  $(a)$   $A_1 := N$  se simule en temps constant par : (fn)  $s_a \text{COPY}(\nu, \alpha_1, \bar{\pi}) s_{a+1}$ .

$(a)$   $N := A_1$  se simule en temps constant par : (fn)  $s_a \text{COPY}(\alpha_1, \nu, \bar{\pi}) s_{a+1}$ .

$(a)$   $A_i := A_1$  se simule en temps constant par : (fn)  $s_a \text{COPY}(\alpha_1, \alpha_i, \bar{\pi}) s_{a+1}$ .

**Construction.**  $(a)$   $A_1 := C_i(A_1, \dots, A_{r_i})$  se simule par l'instruction : (const)  $s_a \alpha_1 \dots \alpha_{r_i} C_i \alpha_1 s_{a+1}$ .

**Destruction.**  $(a)$   $A_1 := \text{dest}_p(A_1)$  se simule par l'instruction : ( $p$ -dest)  $s_a \alpha_1 \alpha_1 s_{a+1}$ .

**HALT.**  $(a)$  HALT ne se simule pas vraiment ; il s'agit plutôt de faire coïncider l'état final de la  $\mathbb{A}$ -RAM avec tous les états correspondant à une instruction HALT.

**Accès mémoire.**  $(a)$   $A_1 := R_{A_1}$  se simule par : (fn)  $s_a \text{ACCESS}(\mu, \mu_1, \mu_2, \mu_3, \mu_4, \alpha_1, \alpha_1, \pi_1, \pi_2, \pi'_1, \pi'_2) s_{a+1}$  en temps  $\mathcal{O}(|\mu| \times |\alpha|)$ .

**Insertion mémoire.** (a)  $R_{A_1} := A_i$  se simule par : (fn)  $s_a \text{INSERT}(\mu, \mu_1, \mu_2, \mu_3, \mu_4, \alpha_1, \alpha_i, \pi_1, \pi_2, \pi'_1, \pi'_2) s_{a+1}$  en temps  $\mathcal{O}(\max(|\alpha|, |\mu|) \times |\mu|)$ .

**Test d'égalité.** (a)  $\text{IF}(A_1 = A_2)\{I(i)\} \text{ELSE} \{I(j)\}$  se simule par : (fn)  $s_a \text{IF}(\alpha_1, \alpha_2, s_i, s_j, \pi_1, \pi_2, \pi'_1, \pi'_2) s_{a+1}$  en temps  $\mathcal{O}(\min(|\alpha_2|, |\alpha_2|))$

**Bilan.** La A-RAM simule lourdement trois des opérations de base de la  $\sigma$ -RAM. Les deux machines coïncident sur les *grosses* classes de complexité ( $P$ ,  $NP$ ,  $EXP$ ...) mais pas sur les classes plus fines ( $\text{DTIME}(n^k)$  pour un  $k$  fixé). ■ □

### Cas particulier : entiers unaires

La  $\sigma$ -RAM permet l'accès à n'importe quel endroit de sa mémoire avec le registre  $A$ . Dans le cas général, la A-RAM doit utiliser des astuces pour reproduire cet accès du mieux possible.

On peut obtenir une légèrement meilleure simulation de la  $\sigma$ -RAM par la A-RAM dans le cas d'une signature  $\sigma = \{0, s(-)\}$ . On note  $\mathbb{N}$  l'algèbre engendrée par  $\sigma$ .

### Une tentative d'amélioration

Enfin, on peut penser à donner un test d'égalité gratuit à la A-RAM. Il accélère la simulation des trois opérations de base de la  $\sigma$ -RAM, mais il reste un coût :

- $A_1 := R_{A_1}$  se simule à présent en temps  $\mathcal{O}(|\mu|)$  ;
- $R_{A_1} := A_i$  se simule à présent en temps  $\mathcal{O}(|\mu|)$  ;
- $\text{IF}(A_1 = A_2)\{I(i)\} \text{ELSE} \{I(j)\}$  se simule à présent en temps  $\mathcal{O}(1)$  ;
- Les autres instructions ne sont pas affectées.

De plus, le test d'égalité gratuit de la N-RAM ne renforce que la simulation du test d'égalité de la  $\{0, s(-)\}$ -RAM. Cette optimisation ne souffrait pas de l'absence d'un test d'égalité gratuit.

La A-RAM garde encore une fois un lourd point faible : la gestion de sa mémoire.

Ici, nous avons utilisé une liste, mais dans le cas d'une algèbre de mots, on pourrait utiliser une mémoire avec embranchements (un terme  $\text{MEM}(-, \dots, -)$ , où chaque emplacement de la mémoire correspond à un constructeur). L'adresse est alors un mot et se traduit comme un ensemble de destructions successives du terme de mémoire, en prenant garde à récupérer le sous-terme correspondant au constructeur détruit.

Sauf qu'il faudrait un accès rapide à n'importe quel sous-terme de la mémoire.

### Problème majeur

La simulation présentée ici de la  $\sigma$ -RAM par la A-RAM n'est pas satisfaisante, car la technique utilisée permet d'avoir la mémoire dans un seul terme, donc de la copier et de la déplacer d'un bloc en une opération, ce qui est beaucoup trop puissant.

De plus, si le test d'égalité est rendu gratuit, on peut faire un test d'égalité entre deux termes de mémoire.

## 1.3 Vers une caractérisation du temps polynomial sur $\sigma$ -RAM

### 1.3.1 Caractérisation du temps linéaire

**Définition 18** (LSRS).

# Appendices

## .1 Simulation de la $\sigma$ -RAM par la $\mathbb{A}$ -RAM

### .1.1 Conventions d'écriture.

Pour des raisons de lisibilité, on écrira des fonctions pour  $\mathbb{A}$ -RAM. Pour faire appel à la fonction, on suivra la convention d'écriture suivante :

(fn)  $s_a$ NOM\_FONCTION(arguments) $s_b$

ce qui se lit : à l'état  $s_a$ , appliquer la fonction puis passer à l'état  $s_b$ . Si la fonction contient un embranchement, comme ce sera le cas pour l'égalité, alors  $s_b$  est inutile mais devrait être précisé comme *bonne pratique* (à l'instar du slash de `<br />` en HTML). De plus, cela permet d'indiquer, à l'écriture d'une fonction, quel est l'état final de la machine qui calcule la fonction.

Deuxièmement, on s'autorisera une boucle *For*, non pas pour rendre le calcul plus puissant, mais bien en tant qu'astuce de lisibilité. Il s'agit en fait d'une boucle *méta* sur l'écriture des commandes.

Par exemple :

```
for p from 1 to r do
| (p-dest)  $s_{a_p} \alpha \pi_p s_{a_{p+1}}$ ;
end
```

doit se comprendre :

```
(1-dest)  $s_{a_1} \alpha \pi_1 s_{a_2}$ ;
:
:
(r-dest)  $s_{a_r} \alpha \pi_r s_{a_{r+1}}$ ;
```

c'est-à-dire comme un enchaînement de commandes.

Troisièmement, pour alléger l'écriture, on autorisera de corriger les labels d'états en notant  $s_a = s_b$  (utile par exemple, lors d'une boucle *méta* sur des indices d'états).

Par exemple, en reprenant notre exemple précédent, les commandes suivantes :

```
for p from 1 to r do
| (p-dest)  $s_{a_p} \alpha \pi_p s_{a_{p+1}}$ ;
end
(correction)  $s_{a_{r+1}} = s_b$ 
```

doit se comprendre :

```
(1-dest)  $s_{a_1} \alpha \pi_1 s_{a_2}$ ;
:
:
(r-dest)  $s_{a_r} \alpha \pi_r s_b$ ;
```

(Remarquons que le dernier état est devenu  $s_b$  au lieu de  $s_{a_{r+1}}$ .)

Enfin, on suppose que les états n'engendrent pas de conflits : si le label de deux états est différent, alors les deux états sont différents. Si un état a le même label dans deux fonctions différentes, alors ces deux états sont différents (chaque fonction est vue comme une boîte noire qui n'agit *que* sur ses paramètres, d'où parfois la nécessité de mettre en paramètre un ou des états).



## .1.2 Ecriture des programmes

```

Input:  $\alpha, \beta, \pi_1, \dots, \pi_r$ 
// Détruit le terme contenu dans  $\alpha$ , stocke ses composantes dans chaque  $\pi_p$ , puis reconstruit le
   terme dans  $\beta$ .
for  $p$  from 1 to  $r$  do
  | ( $p$ -dest)  $s_{a_p} \alpha \pi_1 s_{a_{p+1}}$ ;
end
( $\text{switch}$ )  $s_{a_{r+1}} \alpha s_{b_1} \dots s_{b_k}$ ;
for  $i$  from 1 to  $k$  do
  | ( $\text{const}$ )  $s_{b_i} \pi_1 \dots \pi_{r_i} C_i \beta s_{b_i}$ ;
end

```

**Algorithm 8:** Programme de la fonction  $s_{a_1} \text{COPY}(\alpha, \beta, \bar{\pi}) s_b$ . Algorithme ici.

Pour coder le test d'égalité, on a besoin de quelques sous-fonctions. On fera ici un premier usage du constructeur dédié  $\text{MEM}(-, -, -)$ .

```

Input:  $\mu, \pi_1$ 
// Si  $\mu$  est un terme de mémoire non vide, on extrait la première valeur de cette mémoire en
   laissant le reste inchangé.
if le terme contenu dans  $\mu$  est de la forme  $\text{MEM}(i, A, m)$  then
  | Stocker  $A$  dans  $\pi_1$  ;
  | Remplacer le contenu de  $\mu$  par  $m$  ;
end

```

**Algorithm 9:** Fonction  $s_a \text{EXTRACT\_FIRST}(\mu, \pi_1) s_f$ .

Cette fonction calcule en 3 étapes. Le code se trouve ici :

```

Input:  $\mu, \pi_1$ 
// Si  $\mu$  est un terme de mémoire non vide, on extrait la première valeur de cette mémoire en
   laissant le reste inchangé.
( $\text{switch}$ )  $s_a \mu s_f s_{\text{MEM}}$ ;
(2-dest)  $s_{\text{MEM}} \mu \pi_1 s_b$ ;
(3-dest)  $s_b \mu s_f$ ;

```

**Algorithm 10:** Fonction  $s_a \text{EXTRACT\_FIRST}(\mu, \pi_1) s_f$ . Algorithme ici.

Cette fonction va de pair avec la fonction  $\text{DECOMPOSE}$ .

```

Input:  $\alpha, \pi, \pi', \bar{\pi}$ 
// Détruit le terme contenu dans  $\alpha$ , stocke ses sous-termes dans une seule mémoire  $\pi$  et copie  $\alpha$ 
   dans  $\pi'$ . Les registres  $\bar{\pi}$  sont des registres de travail utiles à la fonction  $\text{COPY}$ .
Selon le constructeur extérieur de  $\alpha$  ;
foreach sous-terme de  $\alpha$  do
  | Stocker le sous-terme dans  $\pi'$ ;
  | Construire dans  $\pi$  le terme  $\text{MEM}(\pi', \pi', \pi)$  ;
end
Copier  $\alpha$  dans  $\pi'$  ;

```

**Algorithm 11:** Fonction  $s_a \text{DECOMPOSE}(\alpha, \pi, \pi', \bar{\pi}) s_c$ .

Cette fonction calcule en  $\leq (k+1)r + 3$  étapes. On peut voir ici un premier usage de la mémoire, comme liste non ordonnée indexée par des termes. Le code de cette fonction se trouve là :

```

Input:  $\alpha, \pi, \pi', \bar{\pi}$ 
// Détruit le terme contenu dans  $\alpha$ , stocke ses sous-termes dans une seule mémoire  $\pi$  et copie  $\alpha$ 
// dans  $\pi'$ . Les registres  $\bar{\pi}$  sont des registres de travail utiles à la fonction COPY.
(switch)  $s_a \alpha s_{a_1} \dots s_{a_k, j}$ ;
for  $i$  from 1 to  $k$  do
  for  $j$  from  $r_i$  to 1 do
    (j-dest)  $s_{a_{i-1}, j} \alpha \pi' s_{b_{i-1}, j}$ ;
    (const)  $s_{b_{i-1}, j} \pi' \pi \text{MEM} \pi s_{a_{i-1}, j+1}$ ;
    (correction)  $s_{a_{i-1}, r_i+1} = s_{a_i, 1}$ ;
  end
  (correction)  $s_{a_k, 1} = s_b$ ;
end
(fn)  $s_b \text{COPY}(\alpha, \pi', \bar{\pi}) s_c$ ;

```

**Algorithm 12:** Fonction  $s_a \text{DECOMPOSE}(\alpha, \pi, \pi', \bar{\pi}) s_c$ .

On peut enfin coder le test d'égalité. On rappelle que l'algorithme se trouve ici.

```

Input:  $\alpha, \beta, s_1, s_0, \pi_1, \pi_2, \pi'_1, \pi'_2$ 
/*  $\alpha, \beta$  : les registres contenant les termes à tester. */
/*  $s_1, s_0$  : place la machine dans l'état  $s_1$  si  $\alpha = \beta$ ,  $s_0$  sinon. */
/*  $\pi_1, \pi_2$  : registres de travail; contiendront respectivement la liste des sous-termes de  $\alpha$  et  $\beta$ 
   qui n'ont pas encore été comparés. */
/*  $\pi'_1, \pi'_2$  : registres de travail; contiendront les sous-termes courants. */
/* */
/* Vérifie si  $\alpha = \beta$  en faisant une analyse inductive sur la construction des termes qu'ils
   contiennent. */

(switch)  $s_a \alpha s_{b_1} \dots s_{b_k}$ ;
for  $i$  from 1 to  $k$  do
  (switch)  $s_{b_i} \beta s_0 \dots s_c \dots s_0$ ;
  // Pour la commande  $s_{b_i}$ , l'état  $s_c$  est en  $i$ -ième position. Ceci permet de vérifier si les
  // constructeurs extérieurs de  $\alpha$  et  $\beta$  sont les mêmes.
  // Si  $C_i$  est un constructeur d'arité 0, alors on écrit  $s_1$  au lieu de  $s_c$  (il n'y a pas de
  // sous-terme à vérifier).
end

(const)  $s_c \in \pi_1 s_{c_1}$ ;
(const)  $s_{c_1} \in \pi_2 s_{c_2}$ ;
// On initialise les registres de travail qui serviront à stocker les sous-termes de  $\alpha$  et  $\beta$ .

(fn)  $s_{c_2}$  DECOMPOSE( $\alpha, \pi'_1, \pi_1$ )  $s_{c_3}$ ;
(fn)  $s_{c_3}$  DECOMPOSE( $\beta, \pi'_2, \pi_2$ )  $s_{d_1}$ ;
// On récupère les sous-termes de  $\alpha$  et  $\beta$ , on les stocke dans  $\pi_1$  et  $\pi_2$ , et on récupère le premier
// sous-terme dans  $\pi'_1$  et  $\pi'_2$ .

(fn)  $s_{d_1}$  EXTRACT_FIRST( $\pi_1, \pi'_1$ )  $s_{d_2}$ ;
(fn)  $s_{d_2}$  EXTRACT_FIRST( $\pi_2, \pi'_2$ )  $s_{e_1}$ ;
// On récupère le premier sous-terme mémorisé dans  $\pi_1, \pi_2$  pour le mettre dans  $\pi'_1, \pi'_2$ .

(switch)  $s_{e_1} \pi'_1 s_{f_1} \dots s_{f_k}$ ;
for  $i$  from 1 to  $k$  do
  (switch)  $s_{f_i} \beta s_0 \dots s_g \dots s_0$ ;
  // De même que pour  $s_{b_i}$ , à la commande  $s_{f_i}$ , l'état  $s_c$  est en  $i$ -ième position.
  // De plus, si le constructeur  $C_i$  est d'arité 0, alors au lieu de  $s_g$ , on écrit  $s_{g_2}$ .
end

(fn)  $s_g$  DECOMPOSE( $\pi'_1, \pi'_1, \pi_1$ )  $s_{g_1}$ ;
(fn)  $s_{g_1}$  DECOMPOSE( $\pi'_2, \pi'_2, \pi_2$ )  $s_{g_2}$ ;
// On récupère les sous-termes de  $\pi'_1$  et  $\pi'_2$ , on les stocke dans  $\pi_1$  et  $\pi_2$ , et on remplace  $\pi'_1$  et  $\pi'_2$ 
// par leur premier sous-terme.

(switch)  $s_{g_2} \pi_1 s_{g_3} s_{g_4}$ ;
(switch)  $s_{g_3} \pi_2 s_1 s_0$ ;
(switch)  $s_{g_4} \pi_2 s_0 s_{d_1}$ ;
// On vérifie les contenus des mémoires temporaires. Si les deux mémoires sont vides
// ( $s_{g_2} \rightarrow s_{g_3} \rightarrow s_1$ ) alors il y a égalité entre les termes de départ (on a fini de tout comparer).
// Si l'une des mémoires est vide et l'autre non ( $s_{g_2} \rightarrow s_{g_3} \rightarrow s_0$  ou  $s_{g_2} \rightarrow s_{g_4} \rightarrow s_0$ ) alors les deux
// termes n'étaient pas égaux. Enfin, si les deux mémoires ont encore des termes à vérifier,
// alors on revient à l'état  $s_{d_1}$ .

```

**Algorithm 13:** Fonction IF( $\alpha, \beta, s_1, s_0, \pi_1, \pi_2, \pi'_1, \pi'_2$ ). Algorithme ici.

```

Input:  $\mu, \mu_1, \mu_2, \mu_3, \mu_4, \alpha, \beta, \pi_1, \pi_2, \pi'_1, \pi'_2$ 
/*  $\mu, (\mu_i)_{i \in 4}$  sont tels que décrits plus haut. */
/*  $\alpha$  est l'indice auquel on veut insérer. */
/*  $\beta$  est la valeur qu'on veut insérer à l'indice  $\alpha$ . */
/*  $\pi_1, \pi_2, \pi'_1, \pi'_2$  sont les registres de travail du test d'égalité. */

// Initialisation de la mémoire
(const) $s_a \varepsilon \mu_4 s_{a_1}$  ;
(1-dest) $s_{a_1} \mu \mu_1 s_{a_2}$  ;
(2-dest) $s_{a_2} \mu \mu_2 s_{a_3}$  ;
(3-dest) $s_{a_3} \mu \mu_3 s_{a_4}$  ;
(const) $s_{a_4} \mu_1 \mu_2 \mu_3 \text{MEM} \mu_3 s_b$  ;

// Première boucle : on avance dans la mémoire en vérifiant à chaque fois si on est au bon
// indice.
(switch) $s_b \mu_3 s_{\varepsilon} s_{\text{MEM}}$  ;

(const) $s_{\varepsilon} \alpha \beta \mu_3 \text{MEM} \mu_3 s_b$  ;

(fn) $s_{\text{MEM}} \text{IF} (\mu_1, \alpha, s_{\text{true}}, s_{\text{false}}, \pi_1, \pi_2, \pi'_1, \pi'_2) s_{s_f}$  ;
(const) $s_{\text{true}} \alpha \beta \mu_3 \text{MEM} \mu_3 s_c$  ;

(const) $s_{\text{false}} \mu_1 \mu_2 \mu_4 \text{MEM} \mu_4 s_{f_1}$  ;
(1-dest) $s_{f_1} \mu \mu_1 s_{f_2}$  ;
(2-dest) $s_{f_2} \mu \mu_2 s_{f_3}$  ;
(3-dest) $s_{f_3} \mu \mu_3 s_b$  ;

// Deuxième boucle : étape inverse : on récupère tout ce qu'on a visité précédemment.
(switch) $s_c \mu_4 s_{c, \varepsilon} s_{c, m}$  ;

(const) $s_{c, \varepsilon} \mu_1 \mu_2 \mu_3 \text{MEM} \mu_3 s_{\text{copie}}$  ;
(1-dest) $s_{c, m} \mu_4 \mu_1 s_{m_1}$  ;
(2-dest) $s_{m_1} \mu_4 \mu_2 s_{m_2}$  ;
(3-dest) $s_{m_2} \mu_4 \mu_4 s_{m_3}$  ;
(const) $s_{m_3} \mu_1 \mu_2 \mu_3 \text{MEM} \mu_3 s_{\text{copie}}$  ;
(fn) $s_{\text{copie}} \text{COPY} (\mu_3, \mu, \mu_1, \mu_2, \mu_4) s_f$  ;

```

**Algorithm 14:** Fonction INSERT ( $\mu, \mu_1, \mu_2, \mu_3, \mu_4, \alpha, \beta, \pi_1, \pi_2, \pi'_1, \pi'_2$ ). Algorithme ici.

```

Input:  $\mu, \mu_1, \mu_2, \mu_3, \mu_4, \alpha, \beta, \pi_1, \pi_2, \pi'_1, \pi'_2$ 
/*  $\mu, (\mu_i)_{i \in 4}$  sont tels que décrits plus haut. */
/*  $\alpha$  est l'indice auquel on veut accéder. */
/*  $\beta$  est le registre dans lequel on veut stocker la valeur d'indice  $\alpha$ . */
/*  $\pi_1, \pi_2, \pi'_1, \pi'_2$  sont les registres de travail du test d'égalité. */

// Initialisation de la mémoire
(const) $s_a \varepsilon \mu_4 s_{a_1}$  ;
(1-dest) $s_{a_1} \mu \mu_1 s_{a_2}$  ;
(2-dest) $s_{a_2} \mu \mu_2 s_{a_3}$  ;
(3-dest) $s_{a_3} \mu \mu_3 s_{a_4}$  ;
(const) $s_{a_4} \mu_1 \mu_2 \mu_3 \text{MEM} \mu_3 s_b$  ;

// Boucle : on avance dans la mémoire en vérifiant à chaque fois si on est au bon indice.
(switch) $s_b \mu_3 s_c s_{\text{MEM}}$  ;

(fn) $s_{\text{MEM}} \text{IF} (\mu_1, \alpha, s_c, s_{\text{false}}, \pi_1, \pi_2, \pi'_1, \pi'_2) s_{s_f}$  ;

(1-dest) $s_{\text{false}} \mu_3 \mu_1 s_{f_1}$  ;
(2-dest) $s_{f_1} \mu_3 \mu_2 s_{f_2}$  ;
(3-dest) $s_{f_2} \mu_3 \mu_3 s_b$  ;

(fn) $s_c \text{COPY} (\mu_2, \beta, \mu_1, \mu_2, \mu_4) s_f$  ;

```

**Algorithm 15:** Fonction ACCESS  $(\mu, \mu_1, \mu_2, \mu_3, \mu_4, \alpha, \beta, \pi_1, \pi_2, \pi'_1, \pi'_2)$ . Algorithme ici.

# Bibliographie

- [1] Stephen Bellantoni and Stephen Cook. A new recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2(2) :97–110, 1992.
- [2] E. Grandjean and T. Schwentick. Machine-independent characterizations and complete problems for deterministic linear time. *SIAM Journal on Computing*, 32(1) :196–230, 2003. cited By 8.
- [3] Daniel Leivant. *Feasible Mathematics II*, chapter Ramified Recurrence and Computational Complexity I : Word Recurrence and Poly-time, pages 320–343. Birkhäuser Boston, Boston, MA, 1995.