

Unification des modèles de calculs caractérisant le temps polynomial

Mémoire de stage

BEURIER Erwan ¹

Master de Logique Mathématique et Fondements de l'Informatique
Université Paris-Diderot - Paris 7, Paris, FRANCE
Année 2015-2016

Soutenu le 1er septembre 2016

1. Adresse mail: erwan.beurier@gmail.com (j'ai bien une adresse institutionnelle mais je n'arrive pas à la rediriger).

Précision et notations

Ce document contient de nombreux liens hypertextes. N'ayant pas trouvé de moyen de les rendre discrets mais visibles, ils sont simplement invisibles. Ne pas hésiter à passer la souris sur des mots comme *ici*, *plus bas*, etc ; ils seront très probablement munis d'un lien hypertexte. Les références à des sections, définitions, théorèmes, etc. sont aussi cliquables. Afin de faciliter la navigation entre programmes et algorithmes dans le premier chapitre, les algorithmes contiennent dans leur légende un lien vers leur programme, et un lien inverse existe aussi sous le programme.

Par abus de notation découlant de la théorie des ensembles, j'écrirai n pour $[0, n - 1]$ voire pour $[1, n]$ quand il n'y aura pas d'ambiguïté. De manière générale, si un terme n ressemble à un entier naturel mais qu'il est mis à la place d'un ensemble (typiquement, dans le domaine de départ ou d'arrivée d'une fonction), il faut le lire comme l'ensemble $[0, n - 1]$.

Table des matières

1	Unification de deux caractérisations de \mathbf{P}	4
1.1	Des caractérisations de \mathbf{P}	4
1.1.1	Notions générales	4
1.1.2	Approche de Cobham	4
1.1.3	Approche de Bellantoni et Cook	5
1.1.4	Approche de Leivant	6
1.1.5	Correspondance entre Bellantoni-Cook et Leivant	7
1.2	Une histoire de RAMs	8
1.2.1	La σ -RAM de Grandjean-Schwentick	8
1.2.2	La \mathbb{A} -RAM de Leivant	9
1.2.3	Comparaison entre les deux RAM	9
2	Deux caractérisations du temps polynomial	19
2.1	Présentation des LSRS	19
2.1.1	Structures de RAM	19
2.1.2	Machine RAM	19
2.1.3	Réductions affines	20
2.2	Le framework algébrique	21
2.2.1	LSRS	21
2.2.2	LRS	22
2.2.3	Théorème principal	22
2.3	Introduction aux LSRS d'arité multiple	23
2.4	Un bon ordre	23
2.4.1	Bon ordre sur les $(\leq a)$ -uplets	23
2.4.2	Propriétés combinatoires	24
2.5	Introduction aux a -LSRS	25
2.6	Lien entre les deux notions de LSRS	26
	Annexes	33
A	Simulation de la σ-RAM par la \mathbb{A}-RAM	34
A.1	Conventions d'écriture	34
A.2	Ecriture des programmes	35
B	Vers une caractérisation algébrique du temps polynomial sur σ-RAM	40
B.1	Première adaptation du résultat de Grandjean et Schwentick	40
B.1.1	n^K -représentable par LRS \Rightarrow calculable en temps $\mathcal{O}(n^K)$	44
B.1.2	Calculable en temps polynomial \Rightarrow LSRS	45

Introduction

Il existe de nombreuses caractérisations de **FP** (notée dans la suite **P** par abus de langage). Parmi les plus fameuses, on peut citer celle de Bellantoni et Cook [1] ou celle de Cobham. Ce sont des caractérisations sur la récurrence, qui permettent de traduire le temps de calcul sur un modèle comme la machine de Turing en contraintes sur les définitions des fonctions, notamment en contrôlant leur récurrence.

Cependant, ces caractérisations ont plusieurs défauts que l'on va essayer de corriger dans ce mémoire :

- Ces caractérisations ne *communiquent* pas, dans le sens où elles passent en général par leur modèle de calcul (machine de Turing ou machine RAM) ; parfois, leurs modèles de calculs sont même spécifiques à la caractérisation. Les buts du chapitre 1 sont premièrement, de trouver des passerelles entre diverses caractérisations sans passer par leur machine respective, et deuxièmement, d'explicitier une simulation entre deux de ces machines ;
- Elles caractérisent **P** globalement, mais ne permettent pas de caractériser des classes plus précises (à savoir, le temps polynomial à polynôme fixé). Le chapitre 2 explore une caractérisation de telles classes en s'inspirant d'un travail effectué sur le temps linéaire [3].

Chapitre 1

Unification de deux caractérisations de \mathbf{P}

1.1 Des caractérisations de \mathbf{P}

Dans cette section, on présente quelques caractérisations connues de \mathbf{P} . Le but de ce premier chapitre va être de lier ces caractérisations de manière plus directe.

1.1.1 Notions générales

Commençons par quelques notions générales.

Définition 1 (Algèbre [6]). 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 σ -algèbre \mathbb{A} est l'ensemble des termes clos constitués uniquement par les symboles de fonctions de σ . Dans ce cas, ces fonctions sont nommées constructeurs.

On note r_i l'arité du constructeur C_i . On suppose que $\min_{i \in [1, k]} (r_i) = 0$ (sinon \mathbb{A} est vide) et $\max_{i \in [1, k]} (r_i) = r > 0$ (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 τ . On peut le définir par induction comme suit :

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

Exemple 2. On distingue deux algèbres particulières :

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

On définit alors la récurrence sur des algèbres de la façon suivante :

Définition 3 (Définition par récurrence sur une algèbre). La fonction $f : \mathbb{A} \rightarrow \mathbb{A}$ est définie par récurrence à partir des fonctions $(g_c)_{c \in \sigma}$ lorsque pour tout constructeur c :

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

1.1.2 Approche de Cobham

On se place 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 4 (Récurrence bornée sur les notations). Une fonction $f \in \mathbf{Cob}$ est définie par récurrence bornée sur les notations à partir de $(g_c)_{c \in \sigma}, h \in \mathbf{Cob}$ lorsque :

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

Définition 5. On note \mathbf{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.

THÉORÈME 6 (Cobham [2]). $\mathbf{Cob} = \mathbf{P}$

L'inclusion $\mathbf{P} \subset \mathbf{Cob}$ consiste à décrire le comportement d'une machine de Turing calculant en temps polynomial avec les fonctions de \mathbf{Cob} ; l'inclusion inverse vient du fait que la borne est toujours polynomiale.

1.1.3 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 7. *Considérons les définitions par récurrence suivantes (dans l'algèbre \mathbb{N}) :*

1. *Addition* : $+(0, y) = y$ et $+(s(x), y) = s(+ (x, y))$;
2. *Multiplication* : $\times (0, y) = 0$ et $\times (s(x), y) = + (x, \times (x, y))$;
3. *Exponentielle* : $\exp(0) = 1$ et $\exp(s(x)) = \exp(x) + \exp(x)$.

Dans ces trois cas, le premier argument est appelé argument de récurrence, puisque c'est celui-ci qui, en quelque sorte, compte le nombre d'étapes de récurrence. Dans l'addition, l'argument $s(x)$ devient x dans l'appel récursif ; idem pour la multiplication. Dans ces deux cas, l'argument de récurrence décroît à chaque appel récursif.

Dans le cas de l'exponentielle par contre, l'argument $s(x)$ devient x dans $\exp(x)$ et $\exp(x)$ en tant qu'argument de récurrence de $+(-, -)$. L'idée de Bellantoni et Cook est d'empêcher qu'un argument de récurrence entraîne la création d'arguments de récurrence de plus en plus gros, ce qui est le cas ici.

Cette idée mène à la séparation des arguments en deux tiers :

- Les arguments *normaux*, qui sont ceux qui ne doivent pas croître trop vite (les premiers arguments dans l'addition et la multiplication) ;
- Les arguments *safe*, qu'on laisse, en quelque sorte, libres de croître.

Au niveau des notations, les arguments *normaux* et *safe* sont séparés par un “;” :

$$f(\underbrace{\bar{x}}_{\text{normaux}} ; \underbrace{\bar{u}}_{\text{safe}})$$

Ensuite, on se place dans une algèbre de mots (tous les constructeurs de l'algèbre sont d'arité au plus 1 ; typiquement, l'algèbre des mots binaires). On utilise uniquement les algèbres de mots car leur récurrence n'engendre qu'un seul calcul à chaque étape, contrairement aux algèbres générales (dites "arborescentes"). Par exemple, une récurrence définie ainsi :

$$f(c(a_1, a_2), \bar{x}) = g_c(f(a_1, \bar{x}), f(a_2, \bar{x}), a_1, a_2, \bar{x})$$

entraîne deux appels récursifs à chaque constructeur, ce qui engendre un nombre exponentiel d'appels au total.

Définition 8 (Composition *safe*). *La fonction f est définie par composition *safe* à partir de $h, g_0, g_1 \in \mathbf{BC}$ lorsque :*

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

Définition 9 (Récurrence *safe*). *La fonction f est définie par récurrence *safe* à partir de $(g_c)_{c \in \sigma} \in \mathbf{BC}$ lorsque pour tout constructeur c :*

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

Définition 10 (Fonctions récursives à arguments normaux [1]). *On note \mathbf{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* : $\text{dest} (; C_i(; x)) = \begin{cases} C_i & \text{si } r_i = 0 \\ x & \text{sinon} \end{cases}$
- la conditionnelle *safe* : $\text{if} (; x, y, z) = \begin{cases} y & \text{si } x \bmod 2 = 0 \\ z & \text{sinon} \end{cases}$

et étant close par schéma de récursion safe et composition safe.

THÉORÈME 11 (Bellantoni et Cook [1]). $\mathbf{BC} = \mathbf{P}$

On démontre en fait $\mathbf{P} \subseteq \mathbf{BC} \subseteq \mathbf{Cob}$. La première inclusion se fait en montrant qu'on peut expliciter les arguments *safe* et *normaux* dans les définitions des fonctions. La deuxième inclusion se trouve en explicitant par induction une borne sur les fonctions de \mathbf{BC} .

1.1.4 Approche de Leivant

Une autre façon de voir les choses est l'approche de Leivant, qui se rapproche de celle de Bellantoni et Cook.

Considérons de nouveau une algèbre quelconque \mathbb{A} de signature $\sigma = \{C_1, \dots, C_k\}$, composée de k symboles de fonctions d'arité respective r_i . Dans la suite, k représentera le nombre de constructeurs dans la signature.

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} , c'est-à-dire, une manière de hiérarchiser les arguments d'une fonction. On notera \mathcal{A} n'importe quel produit cartésien fini de copies de \mathbb{A} . Pour $x \in \mathbb{A}_i$, 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_i de niveau j sera noté C_i^j .

Définition 12 (Composition ramifiée). *Une fonction $f : \mathcal{A} \rightarrow \mathbb{A}_j$ est définie par composition ramifiée à partir de $h : \mathcal{A}' \rightarrow \mathbb{A}_j$ et $(g_i : \mathcal{A} \rightarrow \mathbb{A}_i')_{i \in t}$ lorsque :*

- $\mathbb{A}_1' \times \dots \times \mathbb{A}_t' = \mathcal{A}'$
- $\forall i \quad f(\bar{x}) = h(g_1(\bar{x}), \dots, g_t(\bar{x}))$

Revenons sur la définition de la récurrence ramifiée. Les fonctions g_{c_i} peuvent contenir ou non des appels récursifs à la fonction f . Il s'agit de la notion clé de l'article.

Définition 13 (Argument critique). *Soit f une fonction définie par récurrence à partir des fonctions $(g_{c_i})_{i \in k}$:*

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

Les arguments $f(a_1, \bar{x}), \dots, f(a_{r_i}, \bar{x})$ de la fonction g_{c_i} sont appelés arguments critiques.

Définition 14 (Récurrence ramifiée). *Une fonction $f : \mathbb{A}_m \times \mathcal{A} \rightarrow \mathbb{A}_n$ est définie par récurrence ramifiée à partir des fonctions $(g_{c_i})_{i \in k}$ lorsque :*

- $\forall i \quad g_{c_i} : \mathbb{A}_n^{r_i} \times \mathbb{A}_m^{r_i} \times \mathcal{A} \rightarrow \mathbb{A}_n$
- $\forall i \quad f(c_i^m(a_1, \dots, a_{r_i}), \bar{x}) = g_{c_i}(f(a_1, \bar{x}), \dots, f(a_{r_i}, \bar{x}), \bar{a}, \bar{x})$
- *Si l'une des fonctions de récurrence a un argument critique, alors le tiers de départ m de f doit être strictement supérieur au tiers d'arrivée n : $m > n$; sinon, $m \geq n$.*

Définition 15. *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.*

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 les deux tiers $\mathbb{A}_0, \mathbb{A}_1$.

Un premier résultat sur les tiers 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 (voir lemme 20).

La distinction entre les arguments critiques et les autres permet de définir une notion de degré d'imbrication, qui sert en somme de compteur de tours de boucles.

Définition 16 (Degré d'imbrication de récurrence¹). *Soit $f \in T\text{Rec}(\mathbb{A})$.*

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 $\delta(f) = \max(\delta(g), \delta(h))$;*

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

– 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 $\delta(f) = \max(1 + \delta(g_{\alpha_1}), \dots, 1 + \delta(g_{\alpha_p}), \delta(g_{\beta_1}), \dots, \delta(g_{\beta_q}))$.

Leivant définit aussi une RAM personnalisée pour la construction de termes, voir la définition 23. 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 Grandjean-Schwentick [3] sur des classes plus fines (voir la section 1.2.3 pour la comparaison).

Les principaux résultats de Leivant ne concernent que les algèbres de mots. Voici les résultats de cet article [6] qui nous intéressent dans le cadre de ce mémoire :

THÉORÈME 17. *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 $\delta(f) \leq k$.

On a une caractérisation très fine des classes de complexité à polynôme fixé sur la \mathbb{A} -RAM.

Le deuxième résultat important de [6] est :

THÉORÈME 18. $TRec(\mathbb{A}) = TRec_2(\mathbb{A}) = P$

La récurrence ramifiée engendre P et deux niveaux de ramification suffisent (on retrouve un résultat proche de celui de Bellantoni et Cook).

Le théorème 17 a été prouvé pour une \mathbb{A} -RAM ; l'un des objets de ce stage a été d'obtenir un résultat analogue pour la RAM au sens de Grandjean-Schwentick, voir l'annexe B et le chapitre 2.

1.1.5 Correspondance entre Bellantoni-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 19. *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* $\text{dest}(\cdot; C(\cdot; x))$ 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* $\text{if}(\cdot; x, y, z)$ 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 de composer avec 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 20 (Leivant [6]). *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 18).

Reprenons la démonstration du passage de Leivant à Bellantoni et Cook. Pour des raisons de lisibilité, on écrira x_i pour dire que x est dans le tier i .

Le cas des constructeurs et des projections est trivial.

Si $f(\bar{x}_1, \bar{y}) = g(\bar{x}_1, h_1(\bar{x}), \bar{y}, h_0(\bar{x}, \bar{y}))$ alors :

- Soit $\text{tier}(f) = \text{tier}(g) = 1$, dans ce cas $f(\bar{x}_1) = g(\bar{x}_1, h_1(\bar{x}))$ (car on n'a pas d'argument redondant). Par hypothèse d'induction $h_1(\bar{x}) \equiv h_1(; \bar{x})$ et $g(\bar{x}) \equiv g(; \bar{x})$ d'où $g(\bar{x}_1, h_1(\bar{x})) \equiv g(; \bar{x}, h_1(; \bar{x})) = f(; \bar{x})$.
- 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 σ -RAM de Grandjean-Schwentick

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

Définition 21 (σ -RAM [3]). *Une σ -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 des valeurs entières ; elles s'initialisent à 0.

Un programme de σ -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 σ -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 choisissent (respectivement) la valeur suivante ou l'instruction à lire.

Entrées et sorties. Dans leur article [3], Grandjean et Schwentick indiquent que leur σ -RAM prend en entrée une structure $([1, n], f_1, \dots, f_k, C_1, \dots, C_l)$ où f_1, \dots, f_k sont des fonctions unaires et C_1, \dots, C_l sont des constantes; la σ -RAM 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 $([1, n'], f'_1, \dots, f'_{k'}, C'_1, \dots, C'_{l'})$ stockée de manière similaire dans la σ -RAM.

THÉORÈME 22 ([8]). Une machine de Turing binaire simule une $\{+\}$ -RAM en temps polynomial et une $\{+\}$ -RAM simule une machine de Turing en temps linéaire.

1.2.2 La \mathbb{A} -RAM de Leivant

Soit \mathbb{A} une σ -algèbre.

Définition 23 (\mathbb{A} -RAM [6]). 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 choisie à l'avance.

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

- (const) $s_a \pi_{j_1} \dots \pi_{j_{r_i}} C_i \pi_j s_b$
- (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 π_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 π_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 $*i$ pour le stocker dans π_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 π_j et place la \mathbb{A} -RAM dans l'état s_{b_i} si le constructeur extérieur de $*j$ est C_i . Bien sûr, tous les constructeurs C_i doivent être représentés par un état s_{b_i} .

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

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 π_m .

THÉORÈME 24. 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.

La démonstration se trouve dans [6].

1.2.3 Comparaison entre les deux RAM

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

Simulation d'une \mathbb{A} -RAM dans une σ -RAM

Premières observations. Quelques remarques préalables.

1. Premièrement, la σ -RAM prend en entrée des structures, renvoie une structure en sortie, et manipule 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 σ -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 $DTIME_{\mathbb{A}}(n^k)$ (théorème 17).
2. Deuxièmement, la mémoire de la σ -RAM est infinie et celle de la \mathbb{A} -RAM est finie et ne doit pas dépendre de l'entrée.
3. Troisièmement, la σ -RAM dispose d'un test d'égalité gratuit, ce qui n'est pas le cas de la \mathbb{A} -RAM.
4. Enfin, la \mathbb{A} -RAM a une commande de destruction de terme qui n'a pas réellement d'équivalent dans la σ -RAM.

Afin de contourner le premier problème, on va étendre la définition de la σ -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 σ -RAM pour copier la destruction de terme.

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

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

Définition 25 (σ -RAM constructrice). Une σ -RAM constructrice est un modèle de calcul composé de :

- Des 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 σ -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 $C_i \in \sigma$
- $A_1 := N$
- $A_1 := dest_p(A_1)$ 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 $A_1 := dest_p(A_1)$ 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 \mathbb{A} et \mathbb{N} . 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 σ -RAM sans préciser si elle est constructrice ou non (elle le sera par défaut).

Remarque 26. Cette redéfinition de la σ -RAM a pour but de l'adapter au vocabulaire de la \mathbb{A} -RAM dans un souci de lisibilité et de facilité de démonstration. Cela ne change pas sa puissance d'exécution, du moins dans le premier sens de la simulation : on pourrait simuler les termes dans une σ -RAM avec un système de pointeurs ; par induction, $f(a, b)$ pourrait être représenté par un registre contenant a , un registre contenant b et un registre contenant un label f ainsi que deux pointeurs vers a et b .

Cependant, l'autre simulation (à savoir, σ -RAM simulée par une \mathbb{A} -RAM) aurait été moins lisible.

Enfin, en travaillant avec des termes, les deux machines ont les mêmes opérations de base (à savoir, la construction et destruction de termes), ce qui limite la quantité de codages à faire (notamment pour l'addition).

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

LEMME 27. Une \mathbb{A} -RAM peut être simulée en temps linéaire par une σ -RAM.

Démonstration. Les registres de la \mathbb{A} -RAM seront simplement simulés par les registres de la σ -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 σ -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 $\text{GOTO}(I(j_1), \dots, I(j_t))$ qui permet d'aller aux instructions correspondantes.

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

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

Simulation de la commande (const)

Entrées : Registres sources j_1, \dots, j_{r_i} , registre d'arrivée j , constructeur C_i
// Dans la boucle, on récupère les arguments voulus :
pour t from r_i to 1 **faire**
 $A_1 := f(j_t);$
 $A_1 := R_{A_1};$
 $A_t := A_1;$
fin
// On construit le terme :
 $A_1 := C_i(A_1, \dots, A_{r_i});$
// On stocke le résultat dans $R_{f(j)}$:
 $A_2 := A_1;$
 $A_1 := f(j);$
 $R_{A_1} := A_2;$

Algorithme 1 : Simulation de la commande (const)

Cette simulation se fait en $3r_i + 4$ étapes de calcul. La boucle *for* ici écrite n'est pas une réelle boucle *for*. Il s'agit d'une manière lisible d'écrire des instructions, dont le nombre ne dépend que de r_i .

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

Simulation de la commande (p-dest)

Entrées : Registres i, j
// 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 $R_{f(j)}$:
 $A_2 := A_1;$
 $A_1 := f(j);$
 $R_{A_1} := A_2;$

Algorithme 2 : Simulation de la commande (p-dest)

Cette simulation se fait en 6 étapes de calcul.

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

Rappelons-nous que les états s_{b_i} , dans une A-RAM, correspondent à des numéros d'instruction j_i dans la σ -RAM.

Simulation de la commande (switch)

Entrées : Registre j , instructions $I(j_1), \dots, I(j_k)$

// On détruit entièrement le terme et on récupère chacun des arguments du constructeur extérieur de $R_{f(j)}$. Si l'arité du constructeur est strictement inférieure à p , on assigne une valeur par défaut au registre.

pour p from r to 1 **faire**

$A_1 := f(j)$;

$A_1 := R_{A_1}$;

$A_p := \text{dest}_p(A_1)$;

fin

pour i from 1 to k **faire**

 // 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 :

$A_1 := f(j)$;

$A_1 := R_{A_1}$;

si $A_1 = A_2$ **alors**

 // 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 :

 Aller à l'instruction $I(j_i)$;

sinon

 // 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.

fin

fin

// On construit un nouveau terme en regardant chaque constructeur et on vérifie s'il est égal au terme de $R_{f(j)}$

Algorithme 3 : Simulation de la commande (switch)

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

Bilan. Chaque instruction de la A-RAM peut être simulée en temps constant par une σ -RAM. □

σ -RAM dans A-RAM

LEMME 28. Une σ -RAM fonctionnant en temps polynomial peut être simulée en temps polynomial par une A-RAM.

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

Copie. La A-RAM permet une copie indépendante de la longueur du terme.

Fonction $s_{a_1}\text{COPY}(\alpha, \beta, \bar{\pi})_{s_b}$

/* Récupère chacun des premiers sous-termes de α puis réassemble ces sous-termes dans β . */

Entrées : Registres $\alpha, \beta, \pi_1, \dots, \pi_r$

// Détruit le terme contenu dans α , stocke ses composantes dans chaque π_p .

pour p from 1 to r **faire**

 | Récupérer le p -ième sous-terme de α et le stocker dans π_p ;

fin

Selon le constructeur extérieur de α , utiliser le même constructeur pour construire dans β l'exact même terme, utilisant les mêmes sous-termes stockés dans les π_1, \dots, π_r .

Algorithme 4 : La fonction $s_{a_1}\text{COPY}(\alpha, \beta, \bar{\pi})_{s_b}$. Programme en annexe.

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

Egalité. Pour gérer le premier problème 1, 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. Ce moyen est explicité juste en-dessous.

Fonction $\text{IF}(\alpha, \beta, s_1, s_0, \pi_1, \pi_2, \pi'_1, \pi'_2)$

/* Vérifie si $\alpha = \beta$ en faisant une analyse inductive sur la construction des termes qu'ils contiennent, c'est-à-dire en vérifiant si le constructeur extérieur est le même, puis en déconstruisant le terme et effectuant ces mêmes comparaisons sur les sous-termes. */

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

/* α, β : 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. */

/* π_1, π_2 : registres de travail ; contiendront respectivement la liste des sous-termes de α et β qui n'ont pas encore été comparés. */

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

Copier α dans π'_1 ;

Copier β dans π'_2 ;

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

tant que π_1 et π_2 sont non vides **faire**

si π'_1 et π'_2 ont les mêmes constructeurs extérieurs **alors**

si π'_1 et π'_2 ont des sous-termes **alors**

 Empiler les sous-termes de π'_1 dans π_1 ;

 Empiler les sous-termes de π'_2 dans π_2 ;

fin

 Récupérer le premier élément de π_1 et le stocker dans π'_1 ;

 Récupérer le premier élément de π_2 et le stocker dans π'_2 ;

 Dépiler π_1 ;

 Dépiler π_2 ;

sinon

 Aller à l'état s_1 ;

fin

fin

Aller à l'état s_0 ;

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

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 2, on va supposer que la \mathbb{A} -RAM a le droit d'utiliser deux constructeurs supplémentaires $\text{MEM}(-, -, -)$ et ε . La mémoire de la σ -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 MEM est l'indice du registre (on rappelle que f est la bijection entre les entiers naturels et les termes, cf plus haut), 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 σ -RAM en autorisant un (ou plusieurs) terme(s) de longueur non bornée.

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

Le registre μ_1 contiendra l'indice courant de la mémoire, μ_2 contiendra la valeur du registre associé. μ_3 contiendra la mémoire *suivante* et μ_4 la mémoire *précédente*. Le registre μ est copiée dans μ_3 à l'initialisation de certaines fonctions, puis μ_3 est déroulée, en stockant le contenu passé dans μ_4 .

Une configuration typique à un instant t d'un calcul serait :

- $\mu := \text{MEM}(f(i_1), R_{f(i_1)}, \text{MEM}(\dots, \text{MEM}(f(i_n), R_{f(i_n)}, \varepsilon) \dots))$ (mémoire complète)
- $\mu_1 := f(i_r)$ (pointeur vers une case mémoire)
- $\mu_2 := R_{f(i_r)}$ (contenu de la case mémoire pointée)
- $\mu_3 := \text{MEM}(f(i_{r+1}), R_{f(i_{r+1})}, \text{MEM}(\dots, \text{MEM}(f(i_n), R_{f(i_n)}, \varepsilon) \dots))$ (cases mémoires suivantes)
- $\mu_4 := \text{MEM}(f(i_{r-1}), R_{f(i_{r-1})}, \text{MEM}(\dots, \text{MEM}(f(i_1), R_{f(i_1)}, \varepsilon) \dots))$ (cases mémoires précédentes)

Si l'on veut accéder à la case suivante en mémoire, il faut récupérer les valeurs de μ_1 et μ_2 , les rajouter à μ_4 (ce qui se fait en allongeant le terme de μ_4 : $\mu_4 := \text{MEM}(f(i_{r-1}), R_{f(i_{r-1})}, \mu_4)$), puis raccourcir le terme de μ_3 en récupérant les deux premiers arguments pour remplir μ_1 et μ_2 . Dans la description des algorithmes, on dira qu'on a *dépilé* μ_3 .

On obtient la configuration suivante :

- $\mu := \text{MEM}(f(i_1), R_{f(i_1)}, \text{MEM}(\dots, \text{MEM}(f(i_n), R_{f(i_n)}, \varepsilon) \dots))$ (mémoire complète)
- $\mu_1 := f(i_{r+1})$ (pointeur vers une case mémoire)
- $\mu_2 := R_{f(i_{r+1})}$ (contenu de la case mémoire pointée)
- $\mu_3 := \text{MEM}(f(i_{r+2}), R_{f(i_{r+2})}, \text{MEM}(\dots, \text{MEM}(f(i_n), R_{f(i_n)}, \varepsilon) \dots))$ (cases mémoires suivantes)
- $\mu_4 := \text{MEM}(f(i_r), R_{f(i_r)}, \text{MEM}(\dots, \text{MEM}(f(i_1), R_{f(i_1)}, \varepsilon) \dots))$ (cases mémoires précédentes)

Pour accéder à la case précédente, on effectue la même opération en inversant les rôles de μ_3 et μ_4 .

Pour accéder à une case spécifique de la mémoire, il faut avancer dans la mémoire case par case et effectuer un test d'égalité entre l'indice de la mémoire courante et l'indice auquel on souhaite accéder. La raison à cela est que les termes que l'on utilise ne sont pas ordonnés : si l'on sait à chaque instant quelle case mémoire on est en train de visiter, on ne peut pas savoir quelles sont les cases précédentes et suivantes. Cela impose de réinitialiser les registres $(\mu_i)_{i \in 4}$ à chaque fois qu'on a besoin d'accéder à la mémoire, et de mettre à jour le terme complet de μ à chaque fois qu'on a fini de modifier une case.

A la fin de chaque modification, il faut revenir à la première case mémoire, donc dérouler μ_4 entièrement et le recoller au début de μ_3 , puis copier μ_3 dans μ .

Voici les algorithmes permettant d'effectuer ces opérations en mémoire.

```

Fonction INSERT ( $\mu, \mu_1, \mu_2, \mu_3, \mu_4, \alpha, \beta, \pi_1, \pi_2, \pi'_1, \pi'_2$ )
/* Insère la valeur de  $\beta$  dans la case mémoire d'indice  $\alpha$ . */

Entrées : Registres  $\mu, \mu_1, \mu_2, \mu_3, \mu_4, \alpha, \beta, \pi_1, \pi_2, \pi'_1, \pi'_2$ 
/*  $\mu, (\mu_i)_{i \in 4}$  sont tels que décrits plus haut. */
/*  $\alpha$  est l'indice auquel on veut accéder. */
/*  $\beta$  est le registre récupérant la valeur de la case mémoire d'indice  $\alpha$ . */
/*  $\pi_1, \pi_2, \pi'_1, \pi'_2$  sont les registres de travail nécessaires au 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.
tant que  $\mu_3$  est non-vide faire
    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  $\mu_3$  ;
    si  $\alpha = \mu_1$  alors
        | Sortir de la boucle While ;
    fin
fin
Empiler ( $\mu_1, \beta$ ) dans  $\mu_3$  ;

// Deuxième boucle : étape inverse : on récupère tout ce qu'on a visité précédemment.
tant que  $\mu_4$  est non-vide faire
    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  $\mu_4$  ;
fin
Copier  $\mu_3$  dans  $\mu$  ;

```

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

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


```

Fonction ACCESS ( $\mu, \mu_1, \mu_2, \mu_3, \mu_4, \alpha, \beta, \pi_1, \pi_2, \pi'_1, \pi'_2$ )
/* Copie la valeur de la case mémoire d'indice  $\alpha$  dans  $\beta$ . */

Entrées :  $\mu, \mu_1, \mu_2, \mu_3, \mu_4, \alpha, \beta, \pi_1, \pi_2, \pi'_1, \pi'_2$ 
/*  $\mu, (\mu_i)_{i \in 4}$  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.
tant que  $\mu_3$  est non-vide faire
    Stocker le premier indice de  $\mu_3$  dans  $\mu_1$  ;
    Stocker la première valeur de  $\mu_3$  dans  $\mu_2$  ;
    Dépiler  $\mu_3$  ;
    si  $\alpha = \mu_1$  alors
        | Sortir de la boucle While ;
    fin
fin
Copier  $\mu_2$  dans  $\beta$  ;

```

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

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 σ -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 σ -RAM, on va utiliser une \mathbb{A} -RAM contenant un registre spécial pour chaque accumulateur $(\alpha_i)_{i \in r}$, un registre ν , cinq registres spéciaux dédiés à la gestion de la mémoire $\mu, (\mu_i)_{i \in 4}$, $r + 4$ registres de travail $\bar{\pi}'', \pi_1, \pi_2, \pi'_1, \pi'_2$.

Les états de la \mathbb{A} -RAM seront simplement calqués sur les numéros des instructions du programme de la σ -RAM. Si une instruction de σ -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 σ -RAM (non-constructrice), 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 α_i, ν de la même façon. Si des registres R_i sont initialisés, alors on initialise μ avec un terme mémoire :

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

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 : $(\text{fn}) s_a \text{COPY}(\nu, \alpha_1, \bar{\pi}) s_{a+1}$.

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

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

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

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

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 σ -RAM. Les deux machines coïncident sur les classes de complexité robustes (P , NP , EXP ...) mais pas sur les classes plus fines ($\text{DTIME}(n^k)$ pour un k fixé). \square

Cas particulier : entiers unaires

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

De manière générale, les programmes sont plus simples à écrire car il n'y a qu'un constructeur, et avec une astuce de gestion de mémoire, on arrive à une machine un peu plus rapide. L'astuce consiste à synchroniser la valeur du registre α avec l'indice en mémoire. Si on augmente α de 1, alors on avance d'un pas dans la mémoire.

Cette optimisation repose sur une contrainte : on impose au registre α de la A-RAM de toujours représenter l'indice de la mémoire, puisque c'est le cas dans la σ -RAM. Il faut donc que, à chaque fois que l'on modifie "brutalement" A dans la σ -RAM, par exemple avec l'opération $A := 0$, on se déplace dans la mémoire. Quand on effectue l'opération $A := s(A)$ dans la σ -RAM, on se déplace d'un pas dans la mémoire de la A-RAM.

Voici un petit bilan des gains et pertes :

- L'opération $A := 0$ se simule en temps $\mathcal{O}(|\alpha|)$ au lieu de $\mathcal{O}(1)$;
- L'opération $A := N$ se simule en temps $\mathcal{O}(\max(|\alpha|, |\nu|))$ au lieu de $\mathcal{O}(1)$;
- L'opération $A := R_A$ (accès mémoire) se simule en temps $\mathcal{O}(\max(|\alpha|, |\mu_2|))$ au lieu de $\mathcal{O}(|\mu| \times |\alpha|)$ (on rappelle que μ_2 est la valeur de la case mémoire courante, et μ est le registre contenant toute la mémoire) ;
- L'opération $R_A := B$ (insertion mémoire) se simule en temps $\mathcal{O}(1)$ au lieu de $\mathcal{O}(\max(|\alpha|, |\mu|) \times |\mu|)$
- L'opération $\text{IF}(A_1 = A_2)\{I(i)\} \text{ELSE} \{I(j)\}$ (test d'égalité) se simule en temps $\mathcal{O}(\min(|\alpha|, |\beta|))$, comme avant.
- Le reste des opérations se fait en temps constant.

Le principal gain est pour l'insertion en mémoire. On gagne aussi à l'opération d'accès mémoire, bien que le gain, intuitif, ne soit pas systématique.

Une tentative d'amélioration

Reprenons les algèbres de mots.

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 σ -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.

Conclusion

Les deux modèles de calcul coïncident sur les classes de complexité robustes, mais la simulation de l'un dans l'autre montre un déséquilibre. La σ -RAM est globalement plus rapide que la \mathbb{A} -RAM, et on retrouve naturellement une simulation en temps linéaire de la machine de \mathbb{A} -RAM par la σ -RAM, mais l'inverse ne peut se faire qu'en temps polynomial. Le déséquilibre apparant ici laisse penser qu'on ne pourra pas faire vraiment mieux.

De plus, un tel déséquilibre entraine naturellement que les deux machines définissent des *classes fines* différentes : $\text{DTIME}_{\sigma\text{-RAM}}(n^K)$ et $\text{DTIME}_{\mathbb{A}\text{-RAM}}(n^K)$ sont deux classes de complexité distinctes.

Chapitre 2

Deux caractérisations du temps polynomial

2.1 Présentation des LSRS

Après avoir cherché une passerelle entre deux modèles de calculs, l'objectif du stage était de travailler sur une caractérisation des classes de complexité à polynôme fixé dans une σ -RAM (dans ce chapitre et le suivant, la σ -RAM n'est plus constructive ; on reprend la signature $\sigma = \{+, -, \times\}$).

Une première piste a été de s'inspirer du travail de Grandjean et Schwentick [3] sur le temps linéaire. Cette section de présentation reprend une partie de cet article et l'adapte au temps polynomial. Les preuves n'étant que des adaptations au temps polynomial de celles de cet article, elles ne sont pas incluses dans le mémoire mais se trouvent en annexe de celui-ci.

Définissons d'abord ce qu'on entend par temps $\mathcal{O}(n^K)$. On doit définir les structures de données sur lesquelles on travaille.

2.1.1 Structures de RAM

Définition 29 (RAM-structure). *Soit t un type, c'est-à-dire une signature fonctionnelle ne contenant que des symboles de constantes ou de fonctions unaires.*

Une RAM-structure s de type t est un uplet constitué de :

- $n \in \mathbb{N}$ qui est la taille de la structure ;
- $C \in \mathbb{N}$ pour chaque symbole $C \in t$;
- $f : n \rightarrow \mathbb{N}$ pour chaque symbole $f \in t$.

On notera $s.n, s.C, s.f$ les composantes n, C, f de s (cette notation est à rapprocher de l'accès à un attribut ou à une fonction membre en programmation objet).

On dira que s est c -bornée pour $c \in \mathbb{N}$ lorsque $s.C, s.f(i) < cs.n$ pour tous $C, f \in t$ et $i \in n$.

On peut voir ces structures comme étant des instances d'une *struct* dans le langage C, dont les attributs sont les constantes C et n . Les fonctions f seraient interprétées comme des tableaux de taille n .

Définition 30 (Fonction de RAM). *Soient t_1, t_2 des types.*

Une (t_1, t_2) -fonction de RAM Γ est une fonction telle qu'il existe $c_1, c_2 \in \mathbb{N}$, tels que Γ envoie les structures c_1 -bornées de type t_1 sur des structures c_2 -bornées¹ de type t_2 .

On dit que Γ est polynomiale lorsque $\Gamma(s).n = \mathcal{O}((s.n)^K)$.

Une fonction de RAM n'est qu'une fonction qui calcule une instance de *struct* à partir d'une autre.

2.1.2 Machine RAM

Définition 31 (Temps polynomial). *On définit $DTIME_{RAM}(n^K)$ comme étant l'ensemble des fonctions calculables sur $\{+\}$ -RAM en temps $\mathcal{O}(n^K)$, telles que le nombre de registres utilisés, les valeurs entières manipulées (y compris les adresses de registres) soient bornés par $\mathcal{O}(n^K)$.*

1. On rappelle que "c-borné" ne concerne que la structure par rapport à sa propre taille ; ici on ne compare pas la taille de l'entrée et de la sortie

Pour les besoins de ce mémoire, la machine RAM a pour signature $\{+, -, \times\}$. Elle est intuitivement plus puissante que la $\{+\}$ -RAM utilisée dans l'article [4] qui n'utilise que $+$, mais l'ajout de ces deux opérations ne change pas la puissance de calcul. Autrement dit, une $\{+, -, \times\}$ -RAM travaillant en temps $t(n)$ avec des entiers de taille au plus $t(n)$ peut être simulée par une $\{+\}$ -RAM calculant en temps $t(n)$ et n'utilisant que des entiers de taille au plus $t(n)$. L'algorithme présenté dans [3] revient en fait à écrire une table de multiplication pour tous les entiers, ce qui se fait avant le calcul de la $\{+\}$ -RAM. Les multiplications sont alors remplacées par des accès mémoire, plus rapides.

2.1.3 Réductions affines

On laisse inchangées les notions de *transformations affines*, de *réductions affines*, de *projections affines*. Les théorèmes et lemmes suivants s'y rapportant sont aussi inchangés. Ils permettront de définir des réductions qui *restent* dans $\text{DTIME}_{\text{RAM}}(n^K)$ pour un K fixé.

Définition 32 (Fonction affine non-décroissante). *On appellera fonction affine non-décroissante, ou plus simplement, fonction affine, une fonction A de la forme $A(x_1, \dots, x_k) = a_0 + a_1x_1 + \dots + a_kx_k$, telle que, soit $a_1, \dots, a_k \geq 0$ et $a_0 \in \mathbb{Z}$, soit $a_1, \dots, a_k = 0$ et $a_0 \geq 0$.*

Définition 33 (Transformation affine). *Soit T une fonction de RAM qui envoie des RAM-structures de type t sur des RAM structures de type t' .*

On dit que T est une transformation affine lorsque :

- *Il existe des constantes l, d_1, \dots, d_l et des fonctions affines $\alpha_1, \dots, \alpha_l$ telles que, pour chaque structure de RAM s de type t , on ait $T(s).n = \sum_{i=1}^l d_i \alpha_i(s.n)$;*
- *Pour chaque symbole de constante $C' \neq n$ de t' , il existe une fonction affine $\beta_{C'}$ telle que, pour chaque structure de RAM s de type t , il existe $T(s).C' = \beta_{C'}(s.\bar{C})$, où \bar{C} est le vecteur des constantes de t , n inclus.*
- *Pour chaque symbole de fonction g de t' , chaque $i \leq l$ et chaque $r < d_i$, il existe un symbole de fonction $f_{g,i,r}$ de t , éventuellement le symbole de fonction identité, et une fonction affine $A_{g,i,r}$ tels que pour chaque $i \leq l$, chaque $r < d_i$, chaque structure s , chaque $x < \alpha_i(s.n)$, on ait :*

$$T(s).g \left(r + d_i x + \sum_{j < i} d_j \alpha_j(s.n) \right) = A_{g,i,r}(s.\bar{C}.f_{g,i,r}(x))$$

où $s.f_{g,i,r}(x) = 0$ si $x \geq s.n$.

En résumé, une transformation affine est une fonction qui transforme une structure d'un type en une autre structure d'un autre type, en n'appliquant que des opérations affines sur les éléments de la structure de départ.

Ces transformations sont calculables en temps linéaire et en espace logarithmique par une machine de Turing déterministe.

Il existe plusieurs raffinements de ces transformations.

Définition 34 (Réduction affine). *Soient L_1, L_2 des problèmes de décision (formellement, des ensembles de structures de RAM).*

Une réduction affine de L_1 dans L_2 est une transformation affine telle que pour chaque structure de RAM s , on a $s \in L_1$ ssi $T(s) \in L_2$. Si une telle réduction existe, on note $L_1 \leq_a L_2$.

Ces réductions ont deux avantages : elles sont indépendantes du modèle de calcul utilisé et elles permettent d'effectuer des réductions à l'intérieur d'une classe de complexité restreinte comme peut l'être $\text{DTIME}_{\text{RAM}}(n^K)$, à K fixé.

Dans ce mémoire, nous ne parlerons que de fonctions de RAM et pas de problèmes de décision. On doit raffiner les transformations affines d'une autre manière.

Définition 35 (Projection affine). *Soient t et t' des types de structures de RAM. Une projection affine est une transformation affine $P : (m, s) \mapsto P_m(s)$ où $m \in \mathbb{N}$, s est une structure de RAM de type t et $P_m(s)$ est une structure de RAM de type t' , telle que $P_m(s)$ est définie comme suit :*

- *Pour chaque constante C de t' , et en particulier pour n , les valeurs de $P_m(s).C$ sont définies de l'une des façons suivantes :*
 1. *Il existe un symbole D_C de t tel que, pour chaque structure s de type t et chaque $m \in \mathbb{N}$, on ait $P_m(s).C = s.D_C$;*
 2. *Ou bien, il existe un symbole de fonction f_C de t et une fonction affine $\alpha_C : \mathbb{N} \rightarrow \mathbb{N}$ tels que, pour chaque structure s de type t et chaque $m \in \mathbb{N}$, on ait $P_m(s).C = s.f_C(\alpha_C(m))$.*

- Pour chaque symbole de fonction g de t' , il existe un symbole de fonction f_g de t et une fonction affine $A_g : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ telle que, pour chaque structure s de type t et chaque $m, a \in \mathbb{N}$, on ait : $P_m(s).g(a) = s.f_g(A_g(m, a))$.

Encore une fois, si $A_g(m, a) \geq s.n$, on impose que $s.f_g(A_g(m, a)) = 0$.

Pour résumer, les projections affines opèrent simplement une sélection des valeurs intéressantes qui doivent apparaître dans la structure $P_m(s)$; cette sélection se fait à partir des fonctions affines, lesquelles *savent où regarder* pour trouver les valeurs. De plus, le paramètre m qui apparaît dans la définition est intuitivement la taille de la structure de départ.

2.2 Le framework algébrique

On introduit l'outil qui nous servira à caractériser le temps polynomial.

2.2.1 LSRS

Définition 36 (Application bornée et *equal-predecessor*). Pour $f : n \rightarrow \mathbb{N}$, on définit deux opérations :

- L'application bornée :

$$f[x]_y = \begin{cases} f(x) & \text{si } x < y \\ x & \text{sinon} \end{cases}$$

- L'opération *equal-predecessor* :

$$f^{\leftarrow}(x) = \begin{cases} \max(\{y < x \mid f(x) = f(y)\}) & \text{si un tel } y \text{ existe} \\ x & \text{sinon} \end{cases}$$

Pour $g, g' : n \rightarrow \mathbb{N}$, on combine ces deux opérations pour en créer une troisième, l'opération de récursion :

$$f(x) = g'[g^{\leftarrow}(x)]_x$$

C'est-à-dire : $f(x) = g'(y)$ où y est le plus grand z tel que $g(x) = g(z)$, ou $f(x) = x$ si un tel y n'existe pas.

Définition 37 (LSRS). Soit F un ensemble de symboles de fonctions unaires (dites fonctions de base), soient f_1, \dots, f_k des symboles de fonctions qui n'apparaissent pas dans F . Pour $i \leq k$, notons $F_i = F \cup \{f_1, \dots, f_i\}$.

Un LSRS (Linear Simultaneous Recursion Scheme) S sur f_1, \dots, f_k et F est une suite de k équations $(E_i)_{i \in k}$ dont chacune est de l'une des deux formes suivantes :

- (opération) $f_i(x) = g(x) * g'(x)$ où $g, g' \in F_{i-1}$ et $*$ $\in \{+, -, \times\}$
- (récursion) $f_i(x) = g'[g^{\leftarrow}(x)]_x$ où $g' \in F_k$ et $g \in F_{i-1}$

Remarque 38. Les opérations peuvent varier, il est dit dans [3] que l'on peut choisir n'importe quelle opération binaire calculable en temps linéaire sur machine de Turing. La multiplication est signalée comme n'augmentant pas la puissance de calcul de la RAM, ce qui n'est plus vrai au-delà du temps linéaire.

Pour t un type, on note F_t l'ensemble des symboles de fonctions suivants : $\{1(-), n(-), id(-)\} \cup \{f_C \mid C \in t\} \cup \{f \mid f \in t\}$.

Remarque 39. Soient t un type et S un LSRS pour f_1, \dots, f_k sur F_t . L'entrée d'un LSRS peut être vue comme étant une RAM-structure s de type t , qu'il lit en interprétant les symboles de F_t de la façon suivante :

- $\forall f \in t_1 : f(i) = \begin{cases} s.f(i) & \text{si } i < s.n \\ 0 & \text{sinon} \end{cases}$
- $\forall C \in t_1 : f_C(i) = s.C$
- $1(i) = 1, n(i) = s.n, id(i) = i$

La sortie du LSRS peut aussi être vue comme une nouvelle structure $s' = S(s)$ de type $\{f_1, \dots, f_k\}$.

Définition 40 (RAM n^K -représentée par LSRS). Soient t_1, t_2 des types. Soit Γ une (t_1, t_2) -fonction de RAM.

Soit S un LSRS pour f_1, \dots, f_k sur F_{t_1} .

On dit que Γ est n^K -représentée par S lorsqu'il existe un entier c et une projection affine P tels que, pour chaque structure s c -bornée de type t_1 , S définit des fonctions $f_1, \dots, f_k : c(s.n)^K \rightarrow c(s.n)^K$ telles que $\Gamma(s) = P((s.n)^K, S(s))$ ($S(s)$ est la structure définie par le LSRS).

$$\Gamma : \begin{cases} t_1 & \rightarrow \\ s & \rightarrow \end{cases} \begin{matrix} t_2 \\ P((s.n)^K, S(s)) \end{matrix}$$

Si Γ est n^K -représentée par un LSRS alors on dit que Γ est définissable par LSRS.

La projection affine P permet de redimensionner la sortie du LSRS et d'en extraire les parties importantes pour définir la structure de sortie. Le LSRS définit en quelques sortes des fonctions avec leurs calculs intermédiaires, et la projection affine est un moyen de récupérer les informations utiles du LSRS.

Dans [3], les fonctions du LSRS allaient de $cs.n$ dans $cs.n$. L'adaptation au temps polynomial passe donc principalement par la taille du domaine de départ et d'arrivée des fonctions.

Définition 41 (Définition par cas). *Une fonction f_i est définie par cas à partir de $g_1, g_2, h_1, h_2 \in F_{i-1}$ lorsqu'elle est définie de la manière suivante :*

$$f_i(x) = \begin{cases} g_1(x) & \text{si } h_1(x) \leq h_2(x) \\ g_2(x) & \text{sinon} \end{cases}$$

Démonstration. On peut la construire à partir de la soustraction propre. □

Remarque 42. La comparaison $h_1(x) \leq h_2(x)$ peut être remplacée par $h_1(x) < h_2(x)$, $h_1(x) = h_2(x)$, leur négation. On peut aussi effectuer les opérations booléennes OU et ET.

2.2.2 LRS

Il existe une variante des LSRS qui n'utilise qu'une seule équation mais autorise la composition.

Définition 43 (Terme récursif). *Un terme $\sigma(x)$ est un terme récursif vis-à-vis d'une fonction g et sur le type t lorsque $\sigma(x)$ est de l'une des formes suivantes :*

- $\sigma(x) = C$ où C est une constante de t ;
- $\sigma(x) = f(x)$ où f est une fonction de $t \cup \{1(-), id(-), n(-)\}$;
- $\sigma(x) = g^{\leftarrow}(x - \delta)$ où δ est un entier naturel ;
- $\sigma(x) = g[\sigma'(x)]$ où $\sigma'(x)$ est un terme récursif ;
- $\sigma(x) = f(\sigma'(x))$ où $f \in t$ et $\sigma'(x)$ est un terme récursif ;
- $\sigma(x) = \sigma_1(x) * \sigma_2(x)$ où $\sigma_1(x)$ et $\sigma_2(x)$ sont des termes récursifs, et $*$ $\in \{+, -, \times\}$

Définition 44 (LRS). *Un LRS (Linear Recursion Scheme) pour g et sur t est une équation de la forme $g(x) = \sigma(x)$ où $\sigma(x)$ est un terme récursif vis-à-vis de g et sur le type t .*

Définition 45 (RAM n^K -représentée par LRS). *Soit Γ une fonction de RAM.*

Soit E un LRS :

$$(E) \quad g(x) = \sigma(x)$$

On dit que Γ est n^K -représentée par E lorsqu'il existe un entier c et une projection affine P tels que, pour chaque structure s c -bornée, E définit une fonction $g : c(s.n)^K \rightarrow c(s.n)^K$ telle que $\Gamma(s) = P((s.n)^K, E(s))$ ($E(s)$ est la structure définie par le LRS, elle est de type $\{g\}$).

A l'instar de la définition du LSRS, la principale adaptation au temps polynomial consiste à agrandir la taille des domaines de départ et d'arrivée.

2.2.3 Théorème principal

L'un des principaux théorèmes de l'article [3] s'adapte au temps polynomial :

CONJECTURE 1. *Soit Γ une fonction de RAM. Les propositions suivantes sont équivalentes :*

1. $\Gamma \in DTIME_{RAM}(n^K)$.
2. Γ est n^K -représentée par un LSRS.
3. Γ est n^K -représentée par un LRS.

La démonstration de ce théorème se trouve en annexe. Elle n'est pas présentée dans ce mémoire car la valeur ajoutée par rapport à la démonstration d'origine (donc, pour le temps linéaire) est faible.

2.3 Introduction aux LSRS d'arité multiple

Avant toute chose, il peut être bon d'adapter les LSRS [7] [3]. Pour les besoins du présent chapitre, nous allons ajouter une troisième forme d'opération.

Définition 46 (LSRS). Soit F un ensemble de symboles de fonctions (dites fonctions de base), soient f_1, \dots, f_k des symboles de fonctions qui n'apparaissent pas dans F . Pour $i \leq k$, notons $F_i = F \cup \{f_1, \dots, f_i\}$.

Un LSRS (Linear Simultaneous Recursion Scheme) S sur f_1, \dots, f_k et F est une suite de k équations $(E_i)_{i \in k}$ dont chacune est de l'une des trois formes suivantes :

- (opération) $f_i(x) = g(x) * g'(x)$ où $g, g' \in F_{i-1}$ et $*$ $\in \{+, -, \times\}$
- (récursion) $f_i(x) = g'[g^{\leftarrow}(x)]_x$ où $g' \in F_k$ et $g \in F_{i-1}$
- (composition) $f_i(x) = g'[g(x)]_x$ où $g' \in F_k$ et $g \in F_{i-1}$

Remarque 47. L'ajout de l'opération de composition bornée n'est pas canonique, mais elle facilitera les calculs qui suivront. Elle ne rend pas le LSRS plus puissant [3].

Le but de ce chapitre est d'étendre les LSRS définis dans [3] et utilisés plus haut aux fonctions d'arité > 1 . Remarquons dans un premier temps que si l'on définit un LSRS avec des fonctions d'arité a , où a est le même pour chaque fonction, alors en utilisant l'ordre lexicographique, on peut se ramener à un LSRS équivalent d'arité 1, ce qui n'apporte pas grand-chose par rapport à ce dont nous avons déjà parlé.

Dans la suite, on suppose *a priori* que l'on est dans un cas plus général, où le LSRS généralisé contient des fonctions de plusieurs arités. Soit $a > 1$, on suppose que le LSRS est composé de fonctions f_i d'arité $r_i \leq a$, où $i \in k$, et telles que $r_i = a$ pour au moins un $i \in k$. On qualifie de a -LSRS un tel LSRS. On notera LSRS les systèmes utilisés dans le chapitre précédent, qui sont en fait des 1-LSRS.

Définition 48. Soit a un entier naturel. On appelle $(\leq a)$ -uplet un n -uplet où $n \leq a$.

2.4 Un bon ordre

2.4.1 Bon ordre sur les $(\leq a)$ -uplets

Dans un LSRS, l'ordre est très important ; que ce soit au niveau de l'ordre des fonctions ou des variables. De manière générale, dans un 1-LSRS, $f(x)$ ne peut être défini avec $g(y)$ que si $g(y)$ a été calculé avant, que ce soit parce que $y < x$ ou, si $x = y$, parce que g est définie avant elle dans le LSRS. Pour reproduire l'importance de l'ordre sur des fonctions d'arité multiple, et ne pas trahir la définition originelle du LSRS, on peut utiliser un ordre lexicographique sur les $(\leq a)$ -uplets tels que $\bar{x} <_{\text{naïf}} \bar{y} \Leftrightarrow (|\bar{x}| < |\bar{y}| \text{ ou } \bar{x} <_{\text{lex}} \bar{y})$.

Cela revient à calculer, dans l'ordre, d'abord les fonctions d'arité 1, puis ensuite toutes les fonctions d'arité 2, etc. jusqu'à calculer les fonctions d'arité a . Le problème de cette définition est que les fonctions d'arité a' ne peuvent faire appel qu'aux fonctions d'arité $< a'$, ce qui limite l'usage de cet ordre.

Nous avons choisi un ordre moins naturel, mais qui est un bon ordre, permet de faire des projections (récupérer des éléments d'un a' -uplet, sauf au moins un, et les utiliser comme argument d'une autre fonction), et permet de calculer tour à tour des fonctions d'arité 1, puis 2, etc., puis revenir aux arités 1, 2...

Définition 49 ([5]). On définit l'ordre $<$ sur les $(\leq a)$ -uplets par :

$$\bar{x} < \bar{y} \Leftrightarrow \begin{cases} \max(\bar{x}) < \max(\bar{y}) \\ \text{ou } \max(\bar{x}) = \max(\bar{y}) & \text{et } |\bar{x}| < |\bar{y}| \\ \text{ou } \max(\bar{x}) = \max(\bar{y}) & \text{et } |\bar{x}| = |\bar{y}| \end{cases} \quad \text{et } \bar{x} <_{\text{lex}} \bar{y}$$

Si toutes les arités $\leq a$ ne sont pas permises (par exemple, si notre LSRS ne contient que des fonctions d'arité 1 et 3, mais pas 2), alors on ampute cet ordre des n -uplets correspondant.

Exemple 50. Pour $a = 3$, avec les arités 1, 2, 3 : $(0) < (0, 0) < (0, 0, 0) < (1) < (0, 1) < (1, 0) < (1, 1) < (0, 0, 1) < (0, 1, 0) < (0, 1, 1) < (1, 0, 0) < (1, 0, 1) < (1, 1, 0) < (1, 1, 1) < (2) < (0, 2) < \dots$

Pour $a = 3$, avec les arités 1, 3 : $(0) < (0, 0, 0) < (1) < (0, 0, 1) < (0, 1, 0) < (0, 1, 1) < (1, 0, 0) < (1, 0, 1) < (1, 1, 0) < (1, 1, 1) < (2) < (0, 0, 2) < \dots$

Remarque 51. Quelques remarques informelles :

- Le premier $(\leq a)$ -uplet d'un max m donné est le 1-uplet (m) .
- Le premier $(\leq a)$ -uplet d'une arité r donnée, à un max m donné, est le r -uplet $(0, \dots, 0, m)$.

2.4.2 Propriétés combinatoires

Etudions l'ordre sur les $(\leq a)$ -uplets. Ceci nous servira pour les démonstrations à suivre.

PROPOSITION 52. *Soit \bar{x} un $(\leq a)$ -uplet. On suppose que toutes les arités sont possibles. Notons $m = \max(\bar{x})$ et $r = |\bar{x}|$. Le bon ordre $<$ sur les $(\leq a)$ -uplets vérifie les égalités suivantes :*

1. $\text{card}(\{\bar{y} < \bar{x} \mid \max(\bar{y}) < \max(\bar{x})\}) = \sum_{i=1}^a m^i$;
2. $\text{card}(\{\bar{y} < \bar{x} \mid \max(\bar{y}) = \max(\bar{x}) \wedge |\bar{y}| < |\bar{x}|\}) = \sum_{i=1}^r ((m+1)^i - 1)$;
3. $\text{card}(\{\bar{y} < \bar{x} \mid \max(\bar{y}) = \max(\bar{x}) \wedge |\bar{y}| = |\bar{x}| \wedge \bar{y} <_{\text{lex}} \bar{x}\}) = \sum_{i=1}^r c_i$ où $c_i = x_i \times (m+1)^{r-i}$ si $x_{i-1} = m$ ou $x_{i-2} = m \dots x_1 = m$, et $c_i = (m+1)^{r-i} - m^{r-i}$ sinon.

Démonstration. 1. L'entier a étant fixé, pour un $\max m$ donné, on peut construire un $(\leq a)$ -uplet en choisissant une arité $i \leq a$ et i composantes, toutes des entiers $< m$. Pour i fixé, on peut faire m^i i -uplets, d'où au total $\sum_{i=1}^a m^i$.

2. Pour construire un tel \bar{y} , il faut choisir son arité $i < r$, puis le nombre de composantes que l'on fixe à m ; le reste des composantes est libre et $< m$. On obtient un résultat de la forme $\sum_{i=1}^r \sum_{j=1}^i m^j$ et le résultat après réécriture.

3. D'abord, remarquons que c_i peut être exprimé avec $+$, $-$, \times sous la forme $c_i = (m+1)^{r-i} - \varepsilon_i m^{r-i}$. Par définition de la soustraction propre, ε_i défini² par $1 - ((x_{i-1} + 1) - m) \cdots - ((x_1 + 1) - m)$ vaut 1 si $\forall j \in [1, i-1] x_j < m$ et 0 si l'un des x_j est égal à m .

Expliquons maintenant le résultat par un dénombrement. Observons \bar{x} . Pour construire $\bar{y} < \bar{x}$ de même \max et de même longueur, on peut choisir $y_1 < x_1$, puis laisser les autres coordonnées libres. On a $x_1 \times (m+1)^{r-1}$ façons de construire \bar{y} tel que $\bar{y} <_{\text{lex}} \bar{x}$ et $y_1 < x_1$. Cependant, on ne doit pas compter les r -uplets qui n'affichent pas le $\max m$ dans leurs coordonnées. Comme $y_1 < x_1 \leq m$, m ne peut pas apparaître dans la première coordonnée de \bar{y} . Pour respecter l'ordre que nous avons donné, il faut que le \max apparaisse dans au moins une des $r-1$ autres coordonnées. Il faut donc retirer du compte les r -uplets tels que $y_1 < x_1$ et qui ne contiennent pas le $\max m$ dans leurs $r-1$ autres coordonnées, ce qui donne : $c_1 = x_1 \times ((m+1)^{r-1} - m^{r-1})$.

Une fois que $\sum_{j=1}^{i-1} c_j$ a été calculé pour $i \leq r$, pour construire un $\bar{y} < \bar{x}$ de même \max , de même longueur et ayant fixé les $i-1$ premières coordonnées, on peut fixer $y_i < x_i$ et choisir librement les $r-i$ dernières coordonnées, ce qui se fait dans un premier temps de $x_i \times (m+1)^{r-i}$, sauf qu'il faut s'assurer que le $\max m$ apparaisse bien dans \bar{y} . On a deux possibilités :

- Soit $x_j = m$ pour l'un des $(x_j)_{j < i}$ et dans ce cas il n'y a rien à changer ; on a bien $c_i = x_i \times (m+1)^{r-i}$
- Soit $x_j < m$ pour tous les $(x_j)_{j < i}$ alors il faut retirer du compte les r -uplets tels que $x_j = y_j$ pour $j < i$, $y_i < x_i$ et qui ne contiennent pas le $\max m$ dans leurs $r-i$ autres coordonnées, ce qui donne : $x_i \times ((m+1)^{r-i} - m^{r-i})$.

Finalement, le nombre de r -uplets \bar{y} tels que $\bar{y} < \bar{x}$ et ayant les $j < i$ premières coordonnées en commun est $c_i + \sum_{j=1}^{i-1} c_j$. □

Définition 53. *Le rang de \bar{x} est défini par récurrence :*

- $\text{rang}(0) = 0$
- $\text{rang}(s(\bar{x})) = 1 + \text{rang}(\bar{x})$

C'est le numéro de \bar{x} dans l'ordre $<$.

COROLLAIRE 54.

$$\text{rang}(\bar{x}) = \left(\left(\sum_{i=1}^a m^i \right) + \left(\sum_{i=1}^r ((m+1)^i - 1) \right) + \left(\sum_{i=1}^r c_i \right) \right)$$

où $c_i = x_i \times (m+1)^{r-i}$ si $x_{i-1} = m$ ou $x_{i-2} = m$ ou \dots ou $x_1 = m$, et $c_i = (m+1)^{r-i} - m^{r-i}$ sinon.

2. Attention : ε_i ne prend en compte que les coordonnées x_1 à x_{i-1} .

2.5 Introduction aux a -LSRS

Déroulement du calcul Au lieu d'incrémenter x de façon naturelle en $x+1$, ici, on compte passer de \bar{x} à son successeur dans cet ordre. Notons-le $s(\bar{x})$. Ainsi, le calcul d'un a -LSRS se déroulera de la façon suivante :

- (1) $\bar{x} = (0)$: calcul de toutes les fonctions d'arité 1 en (0) , comme dans un LSRS normal. Dans cette première étape, les fonctions ne peuvent bien sûr pas faire référence aux fonctions d'arité supérieure, car elles ne sont même pas encore définies ; leur évaluation doit respecter les règles du LSRS énoncées plus haut.
- (2) $\bar{x} = (0, 0) = s((0))$: calcul de toutes les fonctions d'arité 2 en $(0, 0)$. Ces fonctions peuvent faire appel aux valeurs des fonctions d'arité 1 en 0. Une fonction f_i d'arité 2 peut aussi faire référence à des fonctions d'arité 2 en $(0, 0)$, à condition qu'elles aient déjà été évaluées ; c'est-à-dire, à condition qu'elles soient définies dans des équations E_j telles que $j < i$;
- (n) $s(\bar{x})$:
 - Si $r = |\bar{x}| = |s(\bar{x})|$ alors on calcule à nouveau les fonctions d'arité r pour $s(\bar{x})$; ces fonctions peuvent faire appel aux résultats des fonctions d'arité plus petite.
 - Si $r = |\bar{x}| < |s(\bar{x})| = r'$ alors on évalue les fonctions d'arité r' en $s(\bar{x})$; idem, ces fonctions peuvent faire appel aux résultats des fonctions d'arité plus petite.
 - Si $|s(\bar{x})| = 1 < a = |\bar{x}|$ alors on évalue les fonctions d'arité 1 en $s(\bar{x})$; ces fonctions d'arité 1 peuvent faire référence aux résultats des fonctions d'arité plus grande, à condition qu'ils aient été calculés au tour précédent.

Une fois l'ordre compris, il faut voir comment interpréter les opérations typiques du LSRS : addition, soustraction, multiplication, récursion et composition.

Définissons le a -LSRS Soit $a \in \mathbb{N}$. Le symbole " $<$ " renvoie selon les cas à l'ordre sur les entiers ou à l'ordre juste au-dessus. Pour des raisons de lisibilité, on notera $\bar{x}' \ll \bar{x}$ pour dire : $\bar{x}' < \bar{x}$, $|\bar{x}'| < |\bar{x}|$ et $\forall j \exists j' x'_j = x_{j'}$; autrement dit : \bar{x}' est obtenu à partir de \bar{x} en récupérant ses composantes, en les mélangeant, en les dupliquant, mais en veillant à ce que $|\bar{x}'| < |\bar{x}|$. Par exemple, pour $\bar{x} = (x_1, x_2, x_3)$, les \bar{x}' pourraient être (x_1, x_2) , (x_3, x_1) ou (x_3, x_3) .

Définition 55 (a -LSRS). Soit F un ensemble de symboles de fonctions de base. Soient f_1, \dots, f_k de nouveaux symboles de fonctions n'apparaissant pas dans F , d'arités respectives $1 \leq r_1 \leq r_2 \leq \dots \leq r_k = a$.

On note $F_i = F \cup \{f_j | r_j = r_i \text{ et } j < i\}$, $F'_i = F \cup \{f_j | r_j = r_i\}$, et $G_i = F \cup \{f_j | r_j < r_i\}$ ³.

Un a -LSRS S sur F et f_1, \dots, f_k est une suite d'équations⁴ E_1, \dots, E_k où chaque E_i est de l'une des formes suivantes :

- (opération) $f_i(\bar{x}) = A * B$ où $*$ $\in \{+, -, \times\}$ et A, B sont de la forme suivante :
 - $g(\bar{x})$, avec $g \in F_i$;
 - $g(\bar{x}')$, avec $g \in G_i$, c et $\bar{x}' \ll \bar{x}$.
- (récursion) $f_i(\bar{x}) = g'[g^{\leftarrow}(\bar{x}')]_{\bar{x}'}$, où $\text{arité}(g) = \text{arité}(g')$ et l'un des deux cas suivants se réalise :
 - Soit $\bar{x}' = \bar{x}$, et dans ce cas $g \in F_i$ et $g' \in F'_i$;
 - Soit $\bar{x}' \ll \bar{x}$ et dans ce cas $g, g' \in G_i$.

Remarque 56. L'opération $g'[g^{\leftarrow}(\bar{x}')]_{\bar{x}'}$ nécessite une brève redéfinition, puisqu'on ne peut plus renvoyer l'unique variable contrairement à précédemment.

$$g'[g^{\leftarrow}(\bar{x}')]_{\bar{x}} = \begin{cases} g'(\bar{y}) & \text{où } \bar{y} = \max(\{\bar{z} < \bar{x} | g(\bar{x}) = g(\bar{z})\}) \text{ si un tel } \bar{y} \text{ existe} \\ x'_i & \text{sinon, où } x'_i \text{ est l'une des coordonnées de } \bar{x}' \end{cases}$$

On impose que la coordonnée renvoyée soit toujours la même au cours du calcul. On pourrait la préciser en ajoutant un label à l'opération, mais dans la suite de ce mémoire, elle ne sera pas cruciale.

Remarque 57. Par défaut, on suppose, une nouvelle fois, que l'ensemble des fonctions de base contient par défaut les symboles $n(-)$ et $1(-)$ pour toutes les arités possibles, ainsi que les $\pi_j^i(-)$, projections récupérant la i -ième composante d'un j -uplet. On peut de plus rajouter $a(-)$, permettant de récupérer a , quoique cette fonction est facilement calculable par un LSRS.

3. F_i est l'ensemble des symboles des fonctions qui ont la même arité que f_i mais qui sont définies avant f_i .

F'_i est l'ensemble des symboles des fonctions qui ont la même arité que f_i , qu'elles soient définies avant ou après f_i .

G_i est l'ensemble des symboles des fonctions d'arité strictement inférieure à celle de f_i .

4. L'ordre entre des fonctions de même arité a de l'importance, mais pas entre des fonctions d'arités différentes. On peut donc considérer que les équations sont ordonnées par l'arité de la fonction qu'elles définissent.

On doit aussi adapter les notions de structures de RAM et de fonctions de RAM. On garde le même a que pour le a -LSRS.

Définition 58 (RAM-structure). *Soit t un a -type, c'est-à-dire une signature fonctionnelle contenant des fonctions d'arité $\leq a$ et dont au moins un symbole est d'arité a .*

Une RAM-structure s de type t est un uplet constitué de :

- $n \in \mathbb{N}$ qui est la taille de la structure ;
- $C \in \mathbb{N}$ pour chaque symbole $C \in t$;
- $f : n \times \dots \times n \rightarrow \mathbb{N}$ pour chaque symbole $f \in t$.

On notera $s.n, s.C, s.f$ les composantes n, C, f de s .

On dira que s est c -bornée pour $c \in \mathbb{N}$ lorsque $s.C, s.f(\bar{x}) < cs.n$ pour tous $C, f \in t$ et $\bar{x} \in n \times \dots \times n$.

Définition 59 (Fonction de RAM). *Soient t_1, t_2 des a_1, a_2 -types.*

Une (t_1, t_2) -fonction de RAM Γ est une fonction telle qu'il existe $c_1, c_2 \in \mathbb{N}$, tels que Γ envoie les structures c_1 -bornées de type t_1 sur des structures c_2 -bornées de type t_2 ⁵.

On dit que Γ est polynomiale lorsque $\Gamma(s).n = \mathcal{O}((s.n)^K)$, et linéaire lorsque $\Gamma(s).n = \mathcal{O}(s.n)$.

Définition 60 (RAM représentée par a -LSRS). *Soient t_1, t_2 respectivement un a_1 -type et un a_2 -type. Soit Γ une (t_1, t_2) -fonction de RAM.*

Soit S un a -LSRS pour f_1, \dots, f_k sur F_{t_1} .

On dit que Γ est représentée par S lorsqu'il existe un entier c et une projection affine P tels que, pour chaque structure s c -bornée, S définit des fonctions $f_1, \dots, f_k : cs.n \times \dots \times cs.n \rightarrow (cs.n)^a$ telles que $\Gamma(s) = P((s.n)^a, S(s))$ ($S(s)$ est la structure définie par le a -LSRS).

A priori, les a -LSRS prennent en entrée des structures dont l'arité maximale est au plus a , car les définitions que l'on a données ne permettent pas d'utiliser des fonctions dont l'arité est plus grande que a .

2.6 Lien entre les deux notions de LSRS

L'idée principale de ce chapitre est de chercher un lien entre les LSRS et les a -LSRS, par le biais du résultat suivant :

THÉORÈME 61. *Soit Γ une (t_1, t_2) -fonction de RAM, où t_1 est un 1-type et t_2 est un a -type.*

Γ est représentable par un a -LSRS ssi Γ est n^a -représentable par un LSRS.

Pour prouver cette conjecture, nous allons avoir besoin de coder l'entrée.

LEMME 62. *Pour tout $a \in \mathbb{N}$, il existe un 1-LSRS S qui, à une variable x , associe le $(\leq a)$ -uplet \bar{x} de rang x .*

Démonstration. Pour simplifier, on va de nouveau se placer dans le cas où toutes les arités $\leq a$ sont représentées⁶. On va se servir du codage du rang donné en proposition 52, montrer qu'on peut le décoder par LSRS.

Calcul du max m . Considérons le LSRS suivant :

$$f_1(x) = \begin{cases} 0 & \text{si } x < a \\ 1 & \text{si } x = a \\ f_1(x-1) & \text{si } x < f_{4a-1}(x-1) \\ f_1(x-1) + 1 & \text{si } x = f_{4a-1}(x-1) \end{cases} \quad (2.1)$$

$$f_{i_1+1}(x) = f_{i_1}(x) \times f_1(x) \quad \text{pour } 1 < i_1 < a \quad (2.2)$$

$$f_{a+1}(x) = f_1(x) + f_2(x) \quad (2.3)$$

$$f_{a+i_2+1}(x) = f_{a+i_2}(x) + f_{i_2+2}(x) \quad \text{pour } 1 < i_2 < a \quad (2.4)$$

$$f_{2a}(x) = f_1(x) + 1 \quad (2.5)$$

$$f_{2a+i_3+1}(x) = f_{2a+i_3}(x) \times f_{2a}(x) \quad \text{pour } 1 < i_3 < a \quad (2.6)$$

$$f_{3a}(x) = f_{2a}(x) + f_{2a+1}(x) \quad (2.7)$$

$$f_{3a+i_4+1}(x) = f_{3a+i_4}(x) + f_{2a+i_4+2}(x) \quad \text{pour } 1 < i_4 < a \quad (2.8)$$

5. On rappelle que " c -borné" ne concerne que la structure par rapport à sa propre taille ; ici on ne compare pas la taille de l'entrée et de la sortie.

6. Si certaines arités ne sont pas représentées, alors il suffit de retirer les quelques équations qui y sont associées et de modifier, le cas échéant, les initialisations de certaines fonctions.

Notons⁷ que $f_1(x-1) = f_1[1^\leftarrow(x)]_x$ et que la distinction de cas ne rend pas le LSRS plus puissant [3].

Expliquons les équations.

L'équation (2.1), calculant $f_1(x)$, est censée calculer m , donc le maximum courant. Sachant cela, les équations suivantes coulent de source :

- Les équations (2.2) (pour $f_{i_1+1}(x)$) calcule m^{i_1} pour $i_1 \in [1, a]$. On a aussi $f_a(x) = m^a$.
- L'équation (2.3) (pour $f_{a+1}(x)$) calcule $\sum_{i=1}^2 m^i$ et (2.4) calcule $\sum_{i=1}^{i_2+2} m^i$ pour $i_2 + 2 \leq a$. On a ainsi $f_{2a-1}(x) = \sum_{i=1}^a m^i$, soit le rang du 1-uplet (m) .
- Les équations suivantes sont sur le même schéma, si ce n'est qu'elles travaillent sur $m+1$ au lieu de m . On a $f_{3a-1}(x) = (m+1)^a$ et $f_{4a-1}(x) = \sum_{i=1}^a (m+1)^i$, donc $f_{4a-1}(x)$ contient le rang du 1-uplet du max suivant, à savoir $(m+1)$.

Ceci étant explicité, étudions la définition de $f_1(x)$.

$f_1(x)$ s'initialise naturellement en 0. On parcourt alors tous les $(\leq a)$ -uplets $(0, \dots, 0)$ jusqu'à l'arité a , donc on énumère les a premiers $(\leq a)$ -uplets, et donc l'élément de rang a a bien un max égal à 1 : $f_1(x)$ passe à 1. Quand le calcul avance, x finit par atteindre le rang du max suivant, que l'on calcule par avance dans $f_{4a-1}(x)$. Une fois que ce rang est atteint, on incrémente $f_1(x)$; le reste des équations se met aussi à jour, ce qui assure que $f_{4a-1}(x)$ est toujours le rang du max suivant. La distinction de cas de $f_1(x)$ est donc bien complète.

Calcul de l'arité courante Le calcul de l'arité du $(\leq a)$ -uplet de rang x s'effectue avec la même idée. Pour des raisons de lisibilité, le système présenté ne sera pas un véritable LSRS mais on laisse le lecteur se convaincre qu'il est facile de réécrire ce système sous une véritable forme de LSRS en s'inspirant de ce qui a été fait au-dessus, quitte à remanier la numérotation des fonctions.

On parlera d'arité "courante" pour parler de l'arité du $(\leq a)$ -uplet de rang x ; l'arité "suivante" est, quand à elle, l'arité du premier $(\leq a)$ -uplet situé après celui de rang x , et qui n'est pas de même arité que celui de rang x . Typiquement, l'arité suivante est soit l'arité du $(\leq a)$ -uplet de rang $x+1$, soit 1.

$$f_{4a}(x) = \begin{cases} 1 & \text{si } x = 0 \\ f_{4a}(x-1) & \text{si } x < f_{8a+2}(x-1) \\ 1 & \text{si } x = f_{4a-1}(x-1) \\ f_{4a}(x-1) + 1 & \text{si } x = f_{8a+2}(x-1) \end{cases} \quad (\text{calcule l'arité courante}) \quad (2.9)$$

$$f_{4a+i_1}(x) = \begin{cases} (m+1)^{i_1} - 1 & \text{si } 1 \leq i_1 \leq f_{4a}(x) \leq a \\ 0 & \text{sinon} \end{cases} \quad \text{pour } 1 \leq i_1 \leq a \quad (2.10)$$

$$\begin{aligned} f_{5a+i_2}(x) &= \sum_{i=1}^{i_2} f_{i+4a}(x) \\ &= \sum_{i=1}^{\min(f_{4a}(x), i_2)} ((m+1)^i - 1) \quad \text{pour } 1 \leq i_2 \leq a \end{aligned} \quad (2.11)$$

$$f_{6a+1}(x) = \begin{cases} f_{4a}(x) + 1 & \text{si } f_{4a}(x) < a \\ 1 & \text{sinon} \end{cases} \quad (\text{calcule l'arité suivante}) \quad (2.12)$$

$$f_{6a+i_3+1}(x) = \begin{cases} (m+1)^{i_3} - 1 & \text{si } 1 \leq i_3 < f_{6a+1}(x) \leq a \\ 0 & \text{si } f_{6a+1}(x) = 1 = i_3 \\ 0 & \text{sinon} \end{cases} \quad \text{pour } 1 \leq i_3 \leq a \quad (2.13)$$

$$f_{7a+i_4+1}(x) = \sum_{i=1}^{i_4} f_{i+6a+1}(x) \quad (2.14)$$

$$\begin{aligned} &= \begin{cases} \sum_{i=1}^{\min(f_{6a+1}(x), i_4)} ((m+1)^i - 1) & \text{si } 1 < f_{6a+1}(x) \\ 0 & \text{sinon} \end{cases} \quad \text{pour } 1 \leq i_4 \leq a \\ f_{8a+2}(x) &= f_{8a+1}(x) + f_{2a-1}(x) \quad (\text{calcule le rang où apparaît l'arité suivante}) \end{aligned} \quad (2.15)$$

On va un peu plus vite : les équations (2.10), (2.11), (2.13) et (2.14) sont des calculs intermédiaires aux résultats suivants : si $f_{4a}(x)$ calcule l'arité courante, alors $f_{6a+1}(x)$ calcule l'arité suivante, et $f_{8a+2}(x)$ calcule le rang du premier

⁷ $1^\leftarrow(x) = \max(0, x-1)$; en effet, si $x = 0$ alors on ne peut pas trouver la valeur précédente qui donne le même résultat, et si $x > 0$, la valeur précédant x qui donne 1 par la fonction $1(-)$ est la valeur précédente, à savoir $x-1$.

$f_{6a+1}(x)$ -uplet. Enfin, $f_{4a}(x)$ vérifie le résultat de $f_{8a+2}(x-1)$ pour s'adapter : si $x = f_{8a+2}(x)$ alors on a atteint le seuil qui détermine qu'on est passé à l'arité suivante. Comme $f_{4a}(x)$ se met à jour, $f_{8a+2}(x)$ se met aussi à jour pour donner le rang du premier terme d'arité suivante, donc x ne dépasse jamais $f_{8a+2}(x)$. Les autres cas de la distinction de cas sont triviaux : on retourne à 1 quand la fonction $f_{4a-1}(x-1)$ (cf. plus haut) indique qu'on a changé de max courant.

Exemple 63. Pour $a = 3$, avec les arités 1, 2, 3 : $(0) < (0, 0) < (0, 0, 0) < (1) < (0, 1) < (1, 0) < (1, 1) < (0, 0, 1) < (0, 1, 0) < (0, 1, 1) < (1, 0, 0) < (1, 0, 1) < (1, 1, 0) < (1, 1, 1) < (2) < (0, 2) < \dots$

On aurait alors :

- $f_{4a}(4) = 1$: le quatrième (≤ 3)-uplet est (1) ;
- $f_{6a+1}(4) = 2$: le (≤ 3)-uplet d'arité différente de 1 est (0, 1) ;
- $f_{8a+2}(4) = 5$: le (≤ 3)-uplet (0, 1) est le successeur de (1).

Calcul des cordonnées. Le LSRS définissant les coordonnées de \bar{x} , élément de rang x dans notre ordre, est un brin complexe.

Faisons quelques observations en premier lieu. Fixons $r = 3$, $m = 2$. Le premier triplet de max m dans l'ordre est $(0, 0, 2)$, et le dernier est $(2, 2, 2)$. Enumérons tous les triplets dans l'ordre lexicographique normal sur $[0, 2]^3$, en faisant ressortir tous ceux qui ne sont pas dans l'ordre lexicographique étendu.

$(0, 0, 2) <_{\text{lex}} (0, 1, 0) <_{\text{lex}} (0, 1, 1) <_{\text{lex}} (0, 1, 2) <_{\text{lex}} (0, 2, 0) <_{\text{lex}} (0, 2, 1) <_{\text{lex}} (0, 2, 2) <_{\text{lex}} (1, 0, 0) <_{\text{lex}} (1, 0, 1) <_{\text{lex}} (1, 0, 2) <_{\text{lex}} (1, 1, 0) <_{\text{lex}} (1, 1, 1) <_{\text{lex}} (1, 1, 2) <_{\text{lex}} (1, 2, 0) <_{\text{lex}} (1, 2, 1) <_{\text{lex}} (1, 2, 2) <_{\text{lex}} (2, 0, 0) <_{\text{lex}} \dots <_{\text{lex}} (2, 2, 2)$

On peut voir l'énumération complète (avec l'ordre lexicographique normal) comme une liste des entiers écrits en base $m+1$, et avec l'ordre lexicographique étendu, comme une liste des entiers écrits en base $m+1$ contenant au moins une fois le chiffre m . On remarque aussi que si un r -uplet est de la forme (x_1, \dots, x_{r-1}, m) mais que le r -uplet suivant dans l'ordre lexicographique ne contient pas le max, alors la dernière coordonnée restera m , et on incrémentera x_{r-1} à la place, jusqu'à ce que $x_{r-1} = m$.

Le calcul crucial porte donc sur la dernière coordonnée et sur la nécessité d'effectuer une simulation du r -uplet suivant.

Considérons que m et r sont fixés. Pour l'instant, on ne considère pas les cas particuliers où l'on change d'arité. De plus, on va numéroter les fonctions non pas par rapport aux LSRS précédents (bien que ceux que nous allons décrire se mettent à la suite des systèmes précédents), mais par rapport aux coordonnées qu'elles représentent.

Supposons que le r -uplet x_1, \dots, x_r est représenté par $f_1(x), \dots, f_r(x)$. La simulation se fait dans les fonctions $f'_1(x), \dots, f'_r(x)$.

La simulation du r -uplet suivant dans l'ordre lexicographique est simple : il s'agit d'incrémenter le dernier chiffre de l'écriture en base $m+1$, et de gérer une retenue. Cela se fait facilement avec le système suivant :

$$f'_r(x) = \begin{cases} f_r(x-1) + 1 & \text{si } f_r(x-1) < m \\ 0 & \text{si } f_r(x-1) = m \end{cases} \quad (2.16)$$

$$f'_{i-1}(x) = \begin{cases} f'_{i-1}(x-1) + 1 & \text{si } f'_i(x) = 0 \text{ et } f'_i(x-1) = m \text{ et } f'_{i-1}(x-1) < m \\ 0 & \text{si } f'_i(x) = 0 \text{ et } f'_i(x-1) = m \text{ et } f'_{i-1}(x-1) = m \\ f'_{i-1}(x-1) & \text{sinon} \end{cases} \quad (2.17)$$

$$\text{future_max}(x) = \max(f'_r(x), \dots, f'_1(x)) \quad (2.18)$$

La fonction \max s'obtient à partir de $\max(a, b) = a + (b - a)$.

Une fois que le r -uplet suivant naïf a été calculé, il est temps de revenir à l'ordre lexicographique étendu :

$$f_r(x) = \begin{cases} f_r(x-1) + 1 & \text{si } f_r(x-1) < m \\ f_r(x-1) & \text{si } f_r(x-1) = m \text{ et "le max n'apparaît pas au tour suivant"} \\ 0 & \text{si } f_r(x-1) = m \text{ et "le max apparaît au tour suivant"} \end{cases} \quad (2.19)$$

$$f_{i-1}(x) = f'_{i-1}(x) \quad (2.20)$$

Expliquons.

Le premier cas de $f_r(x)$ est trivial : si $f_r(x-1) < m$, alors le max est déjà quelque part dans le r -uplet, donc on peut incrémenter sans souci. Si $f_r(x-1) = m$, alors il faut faire attention. L'expression "le max n'apparaît pas au tour suivant" se traduit par : " $\text{future_max}(x) \neq m$ ". Si le max apparaît dans la simulation du tour suivant, alors on peut incrémenter la r -ième composante du r -uplet sans risquer de perdre le max. Si le r -uplet suivant dans l'ordre lexicographique ne contient pas le max, le plus petit r -uplet dans l'ordre lexicographique qui possède le max dans ses composantes est celui avec les mêmes coordonnées que celui simulé, sauf la dernière composante, qui contient le max.

Pour $i \leq r$, le calcul de $f_{i-1}(x)$ est beaucoup plus simple : il reprend dans tous les cas le résultat de la simulation. Le cas le plus pointilleux est celui pour $i = r$. Si $f_r(x) = f_r(x-1) + 1$ ou si le max apparaît dans la simulation, alors on suit l'ordre lexicographique naturel. Sinon, la simulation du tour suivant n'a pas fait apparaître le max. Dans ce cas, le r -uplet suivant dans l'ordre lexicographique étendu a les mêmes composantes que le r -uplet simulé, sauf pour la dernière coordonnée, qui est le max. Ainsi, quel que soit le résultat de la simulation, il est valable pour toutes les coordonnées, sauf la dernière, d'où ce calcul.

Maintenant qu'on a les calculs pour arité et max fixés, il reste à gérer les cas où l'un des deux change. Les cas seront décrits informellement, mais il est facile de les traduire en une distinction de cas complétant celles données précédemment.

- Premièrement, on a toujours accès à l'arité courante et au max courant. On peut aussi détecter quand il y a un changement (en comparant avec la valeur précédente). Ceci facilite les calculs suivants.
- Si l'arité courante n'est pas l'arité maximale, alors on impose que les fonctions "coordonnées" (écrites f_i dans les LSRS précédents) renvoient 0 constamment. Les indices des fonctions "coordonnées" étant fixés, il est facile d'insérer ce cas dans la distinction de cas.
- max m fixé, le premier r -uplet d'une arité r donnée est toujours de la forme $(0, \dots, 0, m)$.
- Le premier $(\leq a)$ -uplet d'un max m donné est toujours (m) .

□

Une fois qu'on a les coordonnées, le deuxième problème qui se pose est l'utilisation des coordonnées partielles, ce que transcrivait la notation $\bar{x}' \ll \bar{x}$: \bar{x}' est obtenu en prenant des coordonnées à \bar{x} , en les mélangeant, éventuellement en dupliquant certaines, tant que $|\bar{x}'| < |\bar{x}|$. Une fois qu'on a le $(\leq a)$ -uplet \bar{x} , il est facile de récupérer \bar{x}' . Mais il faut ensuite convertir \bar{x}' en son rang x' afin que le LSRS d'arité 1 puisse s'en servir.

LEMME 64. *Si la fonction $\bar{x} \mapsto \bar{x}'$, où $\bar{x}' \ll \bar{x}$, est simulable par LSRS, alors la fonction $\bar{x} \mapsto \text{rang}(\bar{x}')$ est simulable par LSRS.*

Démonstration. Il suffit d'exprimer le calcul du rang décrit dans ce corollaire. Le max de \bar{x}' n'est pas forcément le même que celui de \bar{x} , donc il faudra rajouter une fonction qui récupérera son max m' , ainsi qu'une fonction qui renverra son arité r' (le calcul de \bar{x}' par rapport à \bar{x} est toujours le même, donc il suffit de rajouter manuellement l'arité dans le LSRS).

Une fois ces résultats obtenus, le rang de \bar{x}' s'exprime ainsi :

$$\text{rang}(\bar{x}') = \left(\sum_{i=1}^a m'^i \right) + \left(\sum_{i=1}^{r'} ((m' + 1)^i - 1) \right) + \left(\sum_{i=1}^r c_i \right) \quad (2.21)$$

où $c_i = x'_i \times (m' + 1)^{r'-i}$ si $x'_{i-1} = m'$ ou $x'_{i-2} = m' \dots x'_1 = m'$, et $c_i = (m' + 1)^{r'-i} - m'^{r'-i}$ sinon.

Il est facile d'obtenir les puissances successives de m' , $m' + 1$, de les sommer et de calculer les deux premières sommes ; cela a même été fait dans la preuve du lemme 62. Le calcul des c_i n'est pas plus compliqué ; s'inspirer de ce qui a été fait précédemment pour les calculs de puissances. □

LEMME 65. *Supposons que g_1, g_2 sont des symboles de fonctions utilisés ou définies dans un a -LSRS, et sont simulées par \hat{g}_1, \hat{g}_2 dans un LSRS.*

*Alors l'opération $f(\bar{x}) = g_1(\bar{x}_1) * g_2(\bar{x}_2)$ d'un a -LSRS, pour $*$ $\in \{+, -, \times\}$, est simulable par LSRS.*

Démonstration. Tout d'abord, présentons quelques notations. Notons $\text{sub}_j(x)$ l'application qui à un code x associe x_j , code du $(\leq a)$ -uplet $\bar{x}_j \ll \bar{x}$.

Dans la deuxième définition du LSRS, on a rajouté l'opération de composition bornée ; cet ajout facilite le calcul suivant.

Le LSRS suivant ⁸simule $f(\bar{x}) = g_1(\bar{x}_1) * g_2(\bar{x}_2)$:

$$f_1(x) = \hat{g}_1[\text{sub}_1(x)]_x \quad (2.22)$$

$$f_2(x) = \hat{g}_2[\text{sub}_2(x)]_x \quad (2.23)$$

$$f^1(x) = f_1(x) * f_2(x) \quad (2.24)$$

□

8. La numérotation est disjointe de celles précédemment données.

Remarque 66. Avant de traiter l'opération suivante, il faut apporter une précision. Soit le 2-LSRS suivant :

$$f_1(x) = n(x) + 1(x) \quad (2.25)$$

$$f_2(x, y) = f_1(x) + 1(y) \quad (2.26)$$

$$f_3(x, y) = f_1(x) + id(y) \quad (2.27)$$

Supposons-le simulé par un LSRS S sur $\hat{f}_1, \hat{f}_2, \hat{f}_3$ (avec éventuellement d'autres fonctions, mais elles ne sont pas importantes pour la remarque).

L'ordre est : $(0) < (0, 0) < (1) < (0, 1) < (1, 0) < (1, 1) < (2) \dots$. Soit \bar{x} un (≤ 2) -uplet, notons $r(\bar{x})$ son rang. La définition donnée ici du déroulement d'un a -LSRS suppose, en quelque sorte, qu'une fois que $f_1(0)$ est calculé, on avance d'un pas dans l'ordre et $f_1(0)$ reste *figé* en attendant qu'on ait calculé $f_2(0, 0)$ et $f_3(0, 0)$. Dans la simulation par un LSRS, chaque fonction renvoie à chaque tour un résultat ; ici, que devrait renvoyer $\hat{f}_1(r(0, 0))$? Une idée pourrait être simplement de renvoyer son dernier résultat. L'idée est que \hat{f}_1 effectue son calcul sur $r(\bar{x})$ tant que $|\bar{x}| = \text{arité}(f_1)$, puis renvoie son dernier résultat tant que l'arité n'est pas la bonne. Il faut donc que chaque fonction ait accès à l'arité de la fonction qu'elle simule, ce qui peut se faire en rajoutant des fonctions auxiliaires, incluses dans le LSRS.

En conclusion, si $f(\bar{x}) = \sigma(\bar{x})$ est une équation de a -LSRS, avec f d'arité r , alors sa simulation est de la forme :

$$\text{arité}_1(x) = 1(x) \quad (2.28)$$

$$\dots$$

$$\text{arité}_a(x) = \text{arité}_{a-1}(x) + 1(x) \quad (2.29)$$

$$\dots$$

$$\text{a_c}(x) = \text{arité courante} \quad (2.30)$$

$$\dots$$

$$\hat{f}(x) = \begin{cases} \hat{\sigma}(\bar{x}) & \text{si } \text{a_c}(x) = \text{arité}_r(x) \\ \hat{f}(x-1) & \text{sinon} \end{cases} \quad (2.31)$$

Dans la suite, pour des raisons de lisibilité, on évitera ce genre d'écriture, et on notera simplement la simulation sous sa forme $\hat{f}(x) = \hat{\sigma}(\bar{x})$.

LEMME 67. Supposons que g_1, g_2 sont des symboles de fonctions utilisés ou définies dans un a -LSRS, et sont simulées par \hat{g}_1, \hat{g}_2 dans un LSRS.

Alors l'opération $f(\bar{x}) = g_1[g_2^{\leftarrow}(\bar{x}')]_{\bar{x}}$ d'un a -LSRS est simulable par un LSRS.

Démonstration. Pour simplifier, commençons par le cas où $\bar{x}' = \bar{x}$.

La remarque précédente impose de revoir le calcul de $g_1[g_2^{\leftarrow}(\bar{x})]_{\bar{x}}$. Rappelons la définition donnée plus haut :

$$g' [g^{\leftarrow}(\bar{x})]_{\bar{x}} = \begin{cases} g'(\bar{y}) & \text{où } \bar{y} = \max(\{\bar{z} < \bar{x} | g(\bar{z}) = g(\bar{z})\}) \text{ si un tel } \bar{z} \text{ existe} \\ x'_1 & \text{sinon, où } x'_1 \text{ est la première coordonnée de } \bar{x}' \end{cases}$$

Observons cet exemple.

	(0)	(0,0)	(1)	(0,1)	(1,0)	(1,1)	(2)	...
$f_1(x)$	a_0	ND	a_1	ND	ND	ND	a_2	...
$f_2(x)$	b_0	ND	b_1	ND	ND	ND	b_2	...
$f_3(x)$	b_0	ND	b_0	ND	ND	ND	b_0	...
$f_1[f_2^{\leftarrow}(x)]_x$	0	ND	1	ND	ND	ND	2	...
$f_1[f_3^{\leftarrow}(x)]_x$	0	ND	a_0	ND	ND	ND	a_1	...

(ND signifie "Non-Défini".)

Supposons que ce tableau représente les valeurs de quelques fonctions unaires d'un 2-LSRS. Une simulation naïve serait :

\bar{x}	(0)	(0,0)	(1)	(0,1)	(1,0)	(1,1)	(2)	...
x	0	1	2	3	4	5	6	...
$\hat{f}_1(x)$	a_0	a_0	a_1	a_1	a_1	a_1	a_2	...
$\hat{f}_2(x)$	b_0	b_0	b_1	b_1	b_1	b_1	b_2	...
$\hat{f}_3(x)$	b_0	b_0	b_0	b_0	b_0	b_0	b_0	...
$\hat{f}_1 \left[\hat{f}_2^{\leftarrow}(x) \right]_x$	0	0	2	2	2	2	6	...
$\hat{f}_1 \left[\hat{f}_3^{\leftarrow}(x) \right]_x$	0	0	a_0	a_0	a_0	a_0	a_1	...

La simulation en LSRS ne renvoie pas le même résultat par défaut, mais cela peut être réglé rapidement.

Premièrement, $f_1 [f_2^{\leftarrow}(x)]_x$ renvoie une coordonnée en cas d'échec de la recherche, alors que $\hat{f}_1 \left[\hat{f}_2^{\leftarrow}(x) \right]_x$ renvoie le rang de l'entrée. C'est pour ça que les valeurs de sortie n'ont rien à voir. On peut corriger ce bug en rajoutant une nouvelle équation qui va tester si la recherche a échoué :

$$g_1(x) = \text{id} \left[\hat{f}_2^{\leftarrow}(x) \right]_x = \begin{cases} \max_x \left(\left\{ y < x \mid \hat{f}_2(x) = \hat{f}_2(y) \right\} \right) & \text{si de tels } y \text{ existent} \\ x & \text{sinon} \end{cases} \quad (2.32)$$

$$g(x) = \begin{cases} \hat{f}_1 \left[\hat{f}_2^{\leftarrow}(x) \right]_x & \text{si } g_1(x) < x \\ x_i & \text{sinon} \end{cases} \quad (2.33)$$

Deuxièmement, dans la simulation naïve les fonctions renvoient leur dernière valeur quand leur équivalente n'est pas définie. Cette condition permet d'assurer que l'equal-predecessor fonctionne correctement. Ici, l'equal-predecessor parcourt une ligne jusqu'à trouver une case dont la valeur est la même que celle de départ. On parcourt ensuite cette colonne pour arriver à la deuxième fonction de l'equal-predecessor et on regarde quelle valeur on trouve dans cette case ; c'est ce que renvoie l'equal-predecessor. Le fait de garder une fonction constante là où elle n'est pas définie remplit simplement des colonnes, et décale le numéro de colonne visitée, mais le résultat renvoyé est le bon.

Il s'agit plutôt d'une convention à appliquer. Dans le a -LSRS de base, on ne peut pas faire appel à des fonctions à d'autres arités que la leur, donc ça n'est pas gênant dans le cadre d'une simulation.

Enfin, pour le cas général où on ne considère pas $\bar{x} = \bar{x}'$, on se ramène au cas précédent en utilisant $f'(\bar{y}) = g_1 [g_2^{\leftarrow}(\bar{y})]_{\bar{y}}$ et $f(\bar{x}) = f'(\bar{x}')$, où $|\bar{y}| = |\bar{x}'|$.

□

Démonstration. (Du théorème 61)

(Preuve du sens \Rightarrow du théorème 61)

On a donc un LSRS pour extraire les coordonnées de \bar{x} à partir de leur rang x (lemme 62), et on a vu que chaque opération d'un a -LSRS pouvait se simuler par un LSRS.

La taille de sortie du LSRS est égale à $\text{rang}(\bar{n}) = \mathcal{O}(n^a)$, où $\bar{n} = (n, \dots, n)$.

(Preuve du sens \Leftarrow du théorème 61)

Comme nous l'avons fait remarquer dans une remarque précédente, il est possible d'obtenir un a -LSRS dont toutes les fonctions sont d'arité a à partir d'un LSRS d'arité 1. La remarque passait sous silence un détail subtil.

Naïvement, il suffit de considérer qu'un entier $x < n^a$ peut être représenté par un a -uplet \bar{x} où $\text{rang}(\bar{x}) = x$, où le rang considéré est celui de l'ordre $<$ sur les a -uplets (et non pas les $(\leq a)$ -uplets).

Dès lors, les opérations $f(x) = g_1(x) * g_2(x)$ et $f(x) = g_1 [g_2^{\leftarrow}(x)]_x$ se simulent naturellement par : $f(\bar{x}) = g_1(\bar{x}) * g_2(\bar{x})$ et $f(\bar{x}) = g_1 [g_2^{\leftarrow}(\bar{x})]_{\bar{x}}$.

Il y a cependant un problème pour l'opération de composition $f(x) = g_1 [g_2(x)]_x$, qui ne peut pas se simuler telle quelle ; cependant, rappelons qu'elle a été ajoutée à la définition 46 pour faciliter l'écriture, mais elle est simulable par un LSRS à partir des deux autres opérations de base [3] ; il n'y a donc pas besoin de la prendre en compte. □

Conclusion

On a donc deux caractérisations fonctionnelles de \mathbf{P} et de ses sous-classes sur machine σ -RAM. La première caractérisation par LSRS *simple*, et encore plus celle par LSRS à arité multiple, sont des outils théoriques, mais leur utilisation pratique n'est pas aisée.

La caractérisation par LSRS *simple* a pour principal défaut que sa puissance est limitée au moment de l'écriture du LSRS, puisque son temps de calcul est défini à ce moment-là.

La caractérisation par LSRS à arité multiple reproduit ce même défaut, puisque cette fois, la puissance de calcul est limitée au moment du choix des arités utilisées. De plus, l'ordre utilisé, s'il est intéressant théoriquement, n'est en pratique pas simple d'usage, car il n'est pas intuitif de réfléchir à un algorithme avec ce type de déroulement.

Il reste donc à modifier ces caractérisations pour les rendre plus utilisables en pratique.

Annexes

Annexe A

Simulation de la σ -RAM par la \mathbb{A} -RAM

A.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 :

$$(\text{fn}) \ s_a \text{NOM_FONCTION}(\text{arguments})s_b$$

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 `
` 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 :

```
pour  $p$  from 1 to  $r$  faire
|  ( $p$ -dest)  $s_{a_p} \alpha \pi_p s_{a_{p+1}}$ ;
fin
```

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 :

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

doivent 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).

A.2 Ecriture des programmes

```

Fonction  $s_{a_1}\text{COPY}(\alpha, \beta, \bar{\pi})s_b$ 
/* Récupère chacun des premiers sous-termes de  $\alpha$  puis réassemble ces sous-termes dans  $\beta$ . */

Entrées :  $\alpha, \beta, \pi_1, \dots, \pi_r$ 
/* Détruit le terme contenu dans  $\alpha$ , stocke ses composantes dans chaque  $\pi_p$ , puis reconstruit le
   terme dans  $\beta$ . */
pour  $p$  from 1 to  $r$  faire
| ( $p$ -dest)  $s_{a_p}\alpha\pi_1s_{a_{p+1}}$ ;
fin
( $\text{switch}$ )  $s_{a_{r+1}}\alpha s_{b_1} \dots s_{b_k}$ ;
pour  $i$  from 1 to  $k$  faire
| ( $\text{const}$ )  $s_{b_i}\pi_1 \dots \pi_{r_i}C_i\beta s_{b_i}$ ;
fin

```

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

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

```

Fonction  $s_a\text{EXTRACT\_FIRST}(\mu, \pi_1)s_f$ 

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

```

Algorithme 9 : Fonction $s_a\text{EXTRACT_FIRST}(\mu, \pi_1)s_f$.

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

```

Fonction  $s_a\text{EXTRACT\_FIRST}(\mu, \pi_1)s_f$ 

Entrées :  $\mu, \pi_1$ 
/* 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)  $s_{\text{MEM}}\mu\pi_1s_b$ ;
(3-dest)  $s_b\mu\mu s_f$ ;

```

Algorithme 10 : Fonction $s_a\text{EXTRACT_FIRST}(\mu, \pi_1)s_f$. Algorithme plus haut.

Cette fonction va de pair avec la fonction DECOMPOSE .

Fonction $s_a\text{DECOMPOSE}(\alpha, \pi, \pi', \bar{\pi})s_c$

Entrées : $\alpha, \pi, \pi', \bar{\pi}$

/* Détruit le terme contenu dans α , stocke ses sous-termes dans une seule mémoire π et copie α dans π' . Les registres $\bar{\pi}$ sont des registres de travail utiles à la fonction COPY. */

Selon le constructeur extérieur de α ;

pour chaque *sous-terme* de α **faire**

 Stocker le sous-terme dans π' ;

 Construire dans π le terme $\text{MEM}(\pi', \pi', \pi)$;

fin

Copier α dans π' ;

Algorithme 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 :

Fonction $s_a\text{DECOMPOSE}(\alpha, \pi, \pi', \bar{\pi})s_c$

Entrées : $\alpha, \pi, \pi', \bar{\pi}$

/* Détruit le terme contenu dans α , stocke ses sous-termes dans une seule mémoire π et copie α dans π' . Les registres $\bar{\pi}$ sont des registres de travail utiles à la fonction COPY. */

(switch) $s_a\alpha s_{a_1} \dots s_{a_k,j}$;

pour i from 1 to k **faire**

pour j from r_i to 1 **faire**

 (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) $s_{a_{i-1},r_i+1} = s_{a_i,1}$;

fin

 (correction) $s_{a_k,1} = s_b$;

fin

(fn) $s_b\text{COPY}(\alpha, \pi', \bar{\pi})s_c$;

Algorithme 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 plus haut.

```

Fonction IF( $\alpha, \beta, s_1, s_0, \pi_1, \pi_2, \pi'_1, \pi'_2$ )
/* Vérifie si  $\alpha = \beta$  en faisant une analyse inductive sur la construction des termes qu'ils
   contiennent, c'est-à-dire en vérifiant si le constructeur extérieur est le même, puis en
   déconstruisant le terme et effectuant ces mêmes comparaisons sur les sous-termes. */

Entrées :  $\alpha, \beta, s_1, s_0, \pi_1, \pi_2, \pi'_1, \pi'_2$ 
/*  $\alpha, \beta$  : 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)  $s_a \alpha s_{b_1} \dots s_{b_k}$ ;
pour  $i$  from 1 to  $k$  faire
    (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).
fin

(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)  $s_{e_1} \pi'_1 s_{f_1} \dots s_{f_k}$ ;
pour  $i$  from 1 to  $k$  faire
    (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  $s_{g_2}$ .
fin

(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  $s_{d_1}$ .

```

Algorithme 13 : Fonction IF($\alpha, \beta, s_1, s_0, \pi_1, \pi_2, \pi'_1, \pi'_2$). Algorithme ci-dessus.

```

Fonction INSERT ( $\mu, \mu_1, \mu_2, \mu_3, \mu_4, \alpha, \beta, \pi_1, \pi_2, \pi'_1, \pi'_2$ )
/* Insère la valeur de  $\beta$  dans la case mémoire d'indice  $\alpha$ . */

Entrées :  $\mu, \mu_1, \mu_2, \mu_3, \mu_4, \alpha, \beta, \pi_1, \pi_2, \pi'_1, \pi'_2$ 
/*  $\mu, (\mu_i)_{i \in 4}$  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) $s_b \mu_3 s_{\varepsilon} \text{MEM}$  ;

(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) $s_{\text{copie}} \text{COPY} (\mu_3, \mu, \mu_1, \mu_2, \mu_4) s_f$  ;

```

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

```

Fonction ACCESS ( $\mu, \mu_1, \mu_2, \mu_3, \mu_4, \alpha, \beta, \pi_1, \pi_2, \pi'_1, \pi'_2$ )
/* Copie la valeur de la case mémoire d'indice  $\alpha$  dans  $\beta$ . */

Entrées :  $\mu, \mu_1, \mu_2, \mu_3, \mu_4, \alpha, \beta, \pi_1, \pi_2, \pi'_1, \pi'_2$ 
/*  $\mu, (\mu_i)_{i \in 4}$  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) $s_b \mu_3 s_c \text{MEM}$  ;

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

```

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

Annexe B

Vers une caractérisation algébrique du temps polynomial sur σ -RAM

B.1 Première adaptation du résultat de Grandjean et Schwentick

Ici se trouve la démonstration de l'adaptation du résultat de Grandjean et Schwentick, basé uniquement sur l'élargissement des domaines de départ et d'arrivée des fonctions.

Cette partie est une annexe à 2.2.3.

CONJECTURE 2. Soit Γ une fonction de RAM. Les propositions suivantes sont équivalentes :

1. $\Gamma \in DTIME_{RAM}(n^K)$.
2. Γ est n^K -représentée par un LSRS.
3. Γ est n^K -représentée par un LRS.

On a montré $2 \Rightarrow 3$ juste dessous.. La preuve de $3 \Rightarrow 1$ se trouve après. La preuve de $1 \Rightarrow 2$ se trouve plus loin.

CONJECTURE 3 (Passage de LSRS à LRS). Toute fonction de RAM n^K -représentée par LSRS est aussi n^K -représentable par LRS.

Démonstration. Soit Γ une fonction de RAM n^K -représentée par un LSRS S sur F_{t_1} pour f_0, \dots, f_{k-1} , comme dans la définition, et soit P la projection affine associée. Soit s une RAM-structure c -bornée de type t_1 . On note $s' = S(s)$ la structure définie par S avec entrée s , et on note $s'' = \Gamma(s) = P((s.n)^K, s')$. Pour simplifier les notations, on va simplement écrire n au lieu de $s.n$.

L'idée est de coder les fonctions $f_0, \dots, f_{k-1} : cn^K \rightarrow cn^K$ par une unique fonction g . Pour s'assurer que la fonction *Equal-Predecessor* fonctionne correctement, le codage des f_i doit avoir des domaines disjoints et des images disjoints. Cela peut se faire, pour chaque $i < k$, en n'utilisant que des valeurs congrues à i modulo k pour la fonction f_i . Puisqu'on va en avoir besoin pour coder/décoder les opérations, on va aussi coder la fonction $\div k$ dans g .

On va étendre le domaine de g pour encoder $\Gamma(s).f$ pour chaque symbole de fonction f du type de la structure de sortie.

Précisément, g va être définie sur $2kc n^K$ de sorte que :

- pour tout $b \in kc n^K$, on a $g(b) = b \div k$;
- pour tous $a \in cn^K$ et $i \in k$, on a $g(kc n^K + ka + i) = kc n^K + kf_i(a) + i$ (★)

Remarque 68. Dans la démonstration originale, $kc n$ est calculé à la volée dans le LRS. Il s'agit d'une abréviation : $cn = \underbrace{n(y) + \dots + n(y)}_{c \text{ fois}}$, et $kc n = \underbrace{cn + \dots + cn}_{k \text{ fois}}$.

La constante c est toujours une valeur explicite, donnée par définition de la fonction. On peut donc construire ce terme. Idem pour k , qui est le nombre d'équations.

Dans notre cas, il faut autoriser la multiplication pour permettre l'écriture d'un terme analogue : $n^K = \underbrace{n(y) \times \dots \times n(y)}_{K \text{ fois}}$,

puis $cn^K = \underbrace{n^K + \dots + n^K}_{c \text{ fois}}$, et $kc n^K = \underbrace{cn^K + \dots + cn^K}_{k \text{ fois}}$.

L'ajout de la multiplication ne rend pas le LSRS, le LRS et la RAM plus puissants, d'après la partie 3.1 et la proposition 2.1 de [3].

Revenons à la démonstration.

En d'autres termes, pour $b \geq kcn^K$ et $b \bmod k = j$, on a $g(b) = kcn^K + kf_j((b - kcn^K) \div k) + j$. On va définir la valeur de $g(b)$ pour $b \in 2kcn^K$ par distinction de cas sur $b \bmod k$, comme décrit dans (1) à (3) ci-dessous.

1. Renumérotions les symboles de $F_{t_1} = \{f|f \in t_1\} \cup \{f_C|C \in t_1\} \cup \{1(-), n(-), id(-)\}$ de la manière suivante : $\{f^0, \dots, f^{l-1}\}$. On va limiter les occurrences de ces symboles. Premièrement, les symboles f_0, \dots, f_{k-1} sont remplacés par f^l, \dots, f^{l+k-1} respectivement. Ensuite, on va introduire l nouvelles fonctions f_0, \dots, f_{l-1} et l équations pour les définir :

- Si f^i vient de t_1 , alors l'équation associée est $f_i(x) = f^i(x)$;
- Si $f^i = id$, alors l'équation associée est $f_i(x) = x$;
- Si $f^i = f_C$, ou 1 ou n , alors l'équation associée est (respectivement) $f_i(x) = C, 1, n$.

On appellera ces équations des *équations d'entrée* ; elles servent justement à remplacer les entrées du LSRS.

Enfin, on remplace dans le LSRS S tous les anciens symboles de fonctions (ceux de F_{t_1}) par les nouveaux (les $(f_i)_{i \in l+k}$). Après ces remplacements, S ne contient plus aucune référence à F_{t_1} , sauf pour les équations d'entrée.

2. Les équations de S sont combinées en une seule équation $g(y) = \sigma(y)$ comme suit. Le terme $\sigma(y)$ est principalement une distinction de cas dépendant de y et $y \bmod k$.

$$g(y) = \begin{cases} 0 & \text{si } y \leq k-1 \\ g(y-k) + 1 & \text{si } k \leq y < kcn^K \\ \sigma_i(y) & \text{si } kcn^K \leq y \text{ et } y \bmod k = i \end{cases}$$

où $\sigma_i(y)$ est un terme récursif qu'on explicitera tout de suite après.

Notons que les deux premiers cas donnent $g(y) = y \div k$ pour chaque $y < kcn^K$, comme voulu. Cela nous permet d'exprimer $y \bmod k$ pour $kcn^K \leq y < 2kcn^K$.

Ensuite, on a vu que la distinction de cas ne rendait pas le LSRS plus puissant.

Enfin, on décrit la construction des termes de récurrence $\sigma_i(y)$ (on distingue la variable y du terme de récurrence, de la variable x des équations) :

- Si E_i est une équation d'entrée, alors $\sigma_i(y)$ est construit comme suit :
 - Si le terme de droite de l'équation est une constante C (éventuellement 1 ou n), alors $\sigma_i(y) = kcn^K + kC + i$;
 - Si le terme de droite de l'équation est x , alors $\sigma_i(y) = kcn^K + kg(y - kcn^K) + i$;
 - Si le terme de droite de l'équation est $f^i(x)$, alors $\sigma_i(y) = kcn^K + kf^i(g(y - kcn^K)) + i$.
Justifions pourquoi cette définition de $\sigma_i(y)$ est correcte. On le fait pour le troisième cas ; les deux autres sont plus simples. Soit $b = kcn^K + ka + i$ avec $a < cn^K$. Alors $g(b - kcn^K) = g(ka + i) = (ka + i) \div k = a$, et $\sigma_i(b) = kcn^K + kf^i(a) + i$, comme voulu.
- Si E_i est de la forme $f_i(x) = f_j(x) - f_{j'}(x)$, alors $\sigma_i(y)$ est défini par :

$$\sigma_i(y) = (g(y - \delta) - kcn^K - j) - (g(y - \delta') - kcn^K - j') + kcn^K + i$$

où $\delta = i - j$ et $\delta' = i - j'$ et, par définition d'un LSRS, $i > j, j'$. Pour vérifier que cette expression est correcte, soient $a \in cn^K$, $b = kcn^K + ka + i$, où $i < k$. Si $g(b - \delta) = g(kcn^K + ka + j) = kcn^K + kf_j(a) + j$ et $g(b - \delta') = g(kcn^K + ka + j') = kcn^K + kf_{j'}(a) + j'$ alors :

$$(g(b - \delta) - kcn^K - j) - (g(b - \delta') - kcn^K - j') = k(f_j(a) - f_{j'}(a)) + kcn^K + i$$

Ce qui est ce qu'on voulait.

- Si E_i est de la forme $f_i(x) = f_j(x) + f_{j'}(x)$, alors son traitement est similaire au cas précédent, à ceci près qu'il faut que l'addition $x + y$ renvoie 0 si $x + y > cn^K$. On redéfinit alors $\sigma_i(y)$:

$$\sigma_i(y) = \begin{cases} \tau(y) + kcn^K + i & \text{si } \tau(y) < kcn^K \\ kcn^K + i & \text{sinon} \end{cases}$$

où $\tau(y) = (g(b - \delta) - kcn^K - j) - (g(b - \delta') - kcn^K - j')$ avec $\delta = i - j$ et $\delta' = i - j'$.

La vérification se passe de la même manière que précédemment.

- Si E_i est de la forme $f_i(x) = f_{j'} [f_j^{\leftarrow}(x)]_x$, où $j < i$ et on suppose sans perte de généralité que $i \leq j'^1$, alors $\sigma_i(y)$ est définie par :

$$\sigma_i(y) = g[g^{\leftarrow}(y - \delta) + \delta']_y - j' + i$$

où $\delta = i - j$ et $\delta' = j' - j$.

Pour justifier ce remplacement, on doit s'assurer que le codage de plusieurs fonctions en une seule ne cause par d'effets de bord quand on utilise *Equal-Predecessor*. Ce qui est crucial ici, c'est que, pour chaque $i < k$, les valeurs de g qui codent f_i soient congruentes à i modulo k . Pour être plus précis, soit $b = kcn^K + ka + i$, avec $a < cn^K$. Alors $b - \delta = kcn^K + ka + i - (i - j) \stackrel{i > j}{=} kcn^K + ka + j$. On a deux sous-cas à étudier :

- $f_j^{\leftarrow}(a) = a$. Dans ce cas, pour aucun $a' < a$, on n'a $f_j(a') = f_j(a)$, donc il n'y a pas de $a' < a$ pour lequel $g(kcn^K + ka' + j) = g(kcn^K + ka + j)$. La définition de g assure que, pour chaque $e \geq kcn^K$, on a $(g(e) \bmod k = j \Leftrightarrow e \bmod k = j)$ et $\forall e, e' \in 2kn^K$, si $g(e) = g(e')$, alors soit $e, e' < kcn^K$, soit $e, e' \geq kcn^K$. Donc $g^{\leftarrow}(kcn^K + ka + j) = kcn^K + ka + j$ et $g^{\leftarrow}(b - \delta) + \delta' = kcn^K + ka + j' \geq kcn^K + ka + i = b$. Ainsi :

$$\sigma_i(b) = g[g^{\leftarrow}(b - \delta) + \delta']_b - j' + i \quad (\text{B.1})$$

$$= (kcn^K + ka + j') - j' + i \quad (\text{B.2})$$

$$= kcn^K + ka + i \quad (\text{B.3})$$

$$= kcn^K + kf_{j'}[f_j^{\leftarrow}(a)]_a + i \quad (\text{B.4})$$

$$= kcn^K + kf_i(a) + i \quad (\text{B.5})$$

$$= g(b), \quad (\text{B.6})$$

comme voulu.

- $f_j^{\leftarrow}(a) = a'$ pour un certain $a' < a$. Dans ce cas, $g(kcn^K + ka + j) = kcn^K + ka' + j$. En conséquence :

$$g^{\leftarrow}(b - \delta) + \delta' = kcn^K + ka' + j' < kcn^K + ka + i = b \quad (\text{B.7})$$

Donc :

$$\sigma_i(b) = g[g^{\leftarrow}(b - \delta) + \delta']_b - j' + i \quad (\text{B.8})$$

$$= g(kcn^K + ka' + j') - j' + i \quad (\text{B.9})$$

$$= (kcn^K + kf_{j'}(a') + j') - j' + i \quad (\text{B.10})$$

$$= kcn^K + kf_{j'}(a') + j' + i \quad (\text{B.11})$$

$$= kcn^K + kf_{j'}[f_j^{\leftarrow}(a)]_a + i \quad (\text{B.12})$$

$$= kcn^K + kf_i(a) + i \quad (\text{B.13})$$

$$= g(b), \quad (\text{B.14})$$

comme souhaité.

3. Maintenant, on complète le LRS pour g . Pour une question de simplicité, on va supposer que t_2 ne contient que le symbole de constante n et un seul symbole de fonction h . Soient $j < k$ et α une fonction affine tels que, pour toute structure s , on ait $\Gamma(s).n = P((s.n)^K, s').n = s'.f_j(\alpha((s.n)^K))$, et soient $i < k$ et A une fonction affine tels que, pour toute structure s et tout $a < \Gamma(s).n$, on ait $\Gamma(s).h(a) = P((s.n)^K, s').h(a) = s'.f_i(A((s.n)^K, a))$. Ici, on ne fait que donner des noms aux éléments qui permettent de définir la structure $\Gamma(s)$; ces fonctions affines α, A et ces entiers i, j existent par définition d'une fonction n^K -représentable par un LSRS.

On a construit g de telle manière que toutes les valeurs de fonctions $f_i(a)$ sont, en quelque sorte, disponibles dans g , mais on a encore deux problèmes à résoudre. Premièrement, les $f_i(a)$ apparaissent uniquement sous une forme codée; deuxièmement, elles ne forment pas un intervalle contigu mais sont éparpillées modulo k .

1. Si $i > j'$ alors on doit rajouter une nouvelle fonction f_l à S , telle que $l > i$, et définie par la nouvelle équation $f_l(x) = f_{j'}$ et remplacer E_i par $f_i(x) = f_l[f_j^{\leftarrow}(x)]_x$

Il faut donc, avant d'extraire les valeurs à l'aide d'une projection affine bien choisie, décoder les valeurs des fonctions et les ramener dans un même intervalle. Pour ça, on élargit le domaine de g à $(2k+2)cn^K$ et on complète la définition de g :

$$g(y) = \begin{cases} \text{comme avant} & \text{si } y < 2kcn^K \\ g \left[g \left[k(y - 2kcn^K) + kcn^K + i \right]_y - kcn^K \right]_y & \text{si } 2kcn^K \leq y < (2k+1)cn^K \\ g \left[g \left[k(y - (2k+1)cn^K) + kcn^K + j \right]_y - kcn^K \right]_y & \text{si } (2k+1)cn^K \leq y < (2k+2)cn^K \end{cases}$$

Il découle de l'équation (\star) (la définition de g sur kcn^K) et de cette définition que, pour tout $a < cn^K$, on a :

$$g(2kcn^K + a) = g \left[g \left[k((2kcn^K + a) - 2kcn^K) + kcn^K + i \right]_{2kcn^K + a} - kcn^K \right]_{2kcn^K + a} \quad (\text{B.15})$$

$$= g \left[g \left[ka + kcn^K + i \right]_{2kcn^K + a} - kcn^K \right]_{2kcn^K + a} \quad (\text{B.16})$$

$$= g \left[g(ka + kcn^K + i) - kcn^K \right]_{2kcn^K + a} \quad (\text{B.17})$$

$$= g \left[(kcn^K + kf_i(a) + i) - kcn^K \right]_{2kcn^K + a} \quad (\text{B.18})$$

$$= g[kf_i(a) + i]_{2kcn^K + a} \quad (\text{B.19})$$

$$= f_i(a) \quad (\text{B.20})$$

Analysons :

- $B.15 \rightarrow B.16$: par soustraction propre.
- $B.16 \rightarrow B.17$: par définition de l'opérateur *application bornée*.
- $B.17 \rightarrow B.18$: parce que $ka + kcn^K + i < 2kcn^K$ donc on reprend la définition de la fonction g sur l'intervalle $2kcn^K$.
- $B.18 \rightarrow B.19$: par soustraction propre.
- $B.19 \rightarrow B.20$: parce que $f_i(a) < cn^K$ et on a le bon décalage avec i .

De la même manière, on obtient, pour $a < cn^K$, $g((2k+1)cn^K + a) = f_j(a)$.

Maintenant, il est aisé de définir une projection affine P' qui extrait $\Gamma(s)$ à partir de s''' de type $\{g\}$, définie par le LRS :

$$\Gamma(s).n = s'''.g(\alpha(cn^K) + (2k+1)cn^K) \quad (\text{B.21})$$

et :

$$\Gamma(s).h(a) = P(n^K, s').h(a) \quad (\text{B.22})$$

$$= s'.f_i(A(n^K, a)) \quad (\text{B.23})$$

$$= s'''.g(2kcn^K + A(n^K, a)) \quad (\text{B.24})$$

Expliquons. Le résultat de B.21 est bien ce qu'il nous faut, parce que $\alpha' : a \mapsto \alpha(ca) + (2k+1)ca$ est bien une fonction affine telle que $P'(n^K, s').n = s'''.g(\alpha'(n^K))$.

Les calculs pour $\Gamma(s).h(a)$ se justifient comme suit :

- $B.22 \rightarrow B.23$: par définition de A et i .
- $B.23 \rightarrow B.24$: parce qu'on a $f_i(a) = g(2kcn^K + a)$ par construction de g .

Le résultat est de plus exprimé sous la forme demandée : $A' : a, b \mapsto 2kcb + A(b, a)$ est bien une fonction affine telle que $P'(n^K, s').h(a) = s'''.g(A'(n^K, a))$.

□

B.1.1 n^K -représentable par LRS \Rightarrow calculable en temps $\mathcal{O}(n^K)$

CONJECTURE 4. Si une fonction de RAM Γ est n^K -représentable par LRS alors $\Gamma \in DTIME_{RAM}(n^K)$.

Démonstration. Soit Γ une fonction de RAM n^K -représentée par une équation E , et soit P la projection affine correspondante. Pour des raisons de simplicité, on va considérer que le type d'entrée de Γ ne contient qu'un seul symbole de fonction f_{in} . Rappelons-nous que E est une équation de la forme $g(x) = \sigma(x)$, où $\sigma(x)$ est un terme de récursion, et qu'il existe des constantes c et K telles que la sortie $s' = \Gamma(s)$ peut être extraite de $(s.n)^K$ et $g : cn^K \rightarrow cn^K$.

Au lieu de définir formellement la RAM pour Γ , on va donner un algorithme qui pourra facilement être converti en une RAM à plusieurs mémoires, puis en une RAM à une mémoire.

Sa structure de données associée sera constituée de variables p, x et, en plus de la structure d'entrée s et de la structure de sortie s' , de quatre tableaux à une dimension $F_{\text{in}}, G, G_{\text{inverse}}$ et EP .

Dans p , on va stocker $c(s.n)^K$ qui borne le domaine de g , définie par E et s .

Ensuite, on a décrire le sens de $F_{\text{in}}, G, G_{\text{inverse}}$ et EP . Pour des indices plus grands que $s.n$, F_{in} vaut 0 ; pour les indices plus petits que $s.n$, F_{in} contient la valeur de $s.f$. Le calcul principal se fait en p étapes, numérotés de 0 à $p-1 = cn^K - 1$.

Après le tour n^i , on devrait avoir les invariants suivants :

- (a) pour tout $j < p$, $G[j] = \begin{cases} g(j) & \text{si } j \leq i \\ j & \text{si } i < j < p \end{cases}$;
- (b) $EP[j] = g^{\leftarrow}(j)$ pour chaque $j \leq i$;
- (c) pour tout $j < p$, $G_{\text{inverse}}[j] = \begin{cases} \max(\{l \leq i \mid g(l) = j\}) & \text{si un tel } l \text{ existe} \\ p & \text{sinon} \end{cases}$.

Le calcul des valeurs de $G[i]$ est assez immédiat. On associe à chaque terme de récurrence $\sigma(x)$ un terme de programmation $\sigma[x]$ de la façon suivante :

- Si $\sigma(x)$ est $1, n, x$ alors $\sigma[x]$ est $1, n, x$, respectivement ;
- Si $\sigma(x) = g^{\leftarrow}(x - \delta)$ pour un certain δ , alors $\sigma[x] = EP[x - \delta]$;
- Si $\sigma(x) = g[\tau(x)]_x$ pour un certain terme de récurrence $\tau(x)$, alors $\sigma[x] = G[\tau[x]]$;
- Si $\sigma(x) = f_{\text{in}}(\tau(x))$ pour un certain terme de récurrence $\tau(x)$, alors $\sigma[x] = F_{\text{in}}[\tau[x]]$;
- Si $\sigma(x) = \tau_1(x) * \tau_2(x)$ pour des termes de récurrence $\tau_1(x)$ et $\tau_2(x)$, alors $\sigma[x] = \tau_1[x] * \tau_2[x]$.

Au tour i , on calcule $G[i]$ en évaluant $\sigma[i]$. La seule difficulté est de bien évaluer les termes $g^{\leftarrow}(x - \delta)$, qui devrait prendre plus qu'un nombre constant d'étapes pour être évalué, suivant la méthode directe. Au lieu de faire ça, on va utiliser le tableau EP qui contient, après le tour i , la valeur de $g^{\leftarrow}(j)$ pour chaque $j \leq i$. G_{inverse} contient toujours une inverse partielle de g sur $[0, j]$ et est utilisé pour calculer EP .

Voici l'algorithme pour Γ :

```

Entrées : s
// Initializations
p := c(s.n)K ;
EP[0] := 0 ;
pour j = 0 to s.n - 1 faire
  | Fin[j] := s.f(j);
fin
pour j = s.n to p - 1 faire
  | Fin[j] := 0;
fin
pour j = 0 to p - 1 faire
  | G[j] := j;
  | Ginverse[j] := p ;
fin
// Main loop
pour i = 0 to p - 1 faire
  | G[i] = σ[i] ;
  | EP[i] := min ( {i, Ginverse[G[i]]} ) ;
  | Ginverse[G[i]] := i ;
fin
Output : Compute s'' by applying the affine projection P(s.n)K to the structure s' = (p, G)

```

On montre par induction que les invariants (a), (b), (c) sont maintenus pendant chaque étape de calcul. Bien évidemment, les invariants sont respectés après l'initialisation, c'est-à-dire, avant le tour 0. Pour l'induction, supposons que les invariants (a), (b), (c) sont respectés *avant* le tour $i \in p$. On montre qu'ils sont valides *après* le tour i , et donc avant le tour $i + 1$.

- On a $G[i] = g(i)$ parce que, par induction, $G[j] = g[j]_i$ pour tout $j \in p$, et $EP[j] = g^{\leftarrow}(j)$ pour tout $j \in i$; ainsi, l'opérateur d'application bornée et la fonction *Equal-Predecessor* sont évaluées correctement.
- L'assignation $EP[i] := \min(\{i, G_{\text{inverse}}[G[i]]\})$ implique que $EP[i] = g^{\leftarrow}(i)$ par induction.
- Il est immédiat de voir que (c) est maintenu par l'assignation $G_{\text{inverse}}[G[i]] := i$.

Donc (a), (b) et (c), et en particulier (a), sont maintenus, donc le programme est correct.

Le nombre d'étapes pour évaluer un terme de récurrence est linéaire en la longueur du terme. Comme le terme de récurrence de E est fixé, il s'agit d'un nombre constant d'étapes. Ainsi, $G[i]$ n'a besoin que d'un nombre constant d'étapes, de sorte que le programme tourne en temps $\mathcal{O}(p) = \mathcal{O}(n^K)$. □

B.1.2 Calculable en temps polynomial \Rightarrow LSRS

CONJECTURE 5. Si une fonction de RAM Γ est calculable en temps $\mathcal{O}(n^K)$ sur une $\{+, -, \times\}$ -RAM M , alors Γ est n^K -représentable par un LSRS.

Démonstration. Soit Γ une fonction de RAM. Pour des raisons de simplicité, on va supposer que les types d'entrée et de sortie de la fonction n'ont qu'un seul symbole de fonction f . Soit M une RAM calculant Γ comme stipulé dans les hypothèses. Soit c tel que le temps de calcul est borné par cn^K sur des entrées $c(\Gamma)$ -bornées de taille n .

On va construire un LSRS qui utilise des fonctions I, A, B, R_A, N , qui décrivent l'état courant de la RAM *avant* chaque étape x , de la façon suivante :

- $I(x)$ contient le numéro courant d'instruction du programme de la RAM ;
- $A(x), B(x), N(x)$ contiennent les valeurs des registres A, B, N de M ;
- $R_A(x)$ contient la valeur du registre dont l'adresse est actuellement dans le registre A .

Par commodité, on va aussi utiliser des fonctions I', A', B', R'_A, N' , qui décrivent l'état de la RAM *après* l'étape x .

En définissant $I(x)$ par distinction de cas, une bonne partie de la simulation de la RAM est immédiate. Par exemple, si M exécute $A := A + B$ alors les équations doivent forcer $A'(x) = A(x) + B(x)$. Le principal problème réside dans l'instruction $A := R_A$, qui récupère le contenu d'un registre. Notons que les fonctions définies dans S n'encodent pas explicitement les valeurs de tous les registres de M à chaque étape t mais seulement le contenu du registre dont l'indice est contenu dans A . Pour obtenir la valeur de $R_A(t)$, on doit trouver le dernier instant avant t auquel le registre A avait la valeur $A(t)$. Si un tel instant n'existe pas, on doit se référer à l'entrée. Rappelons que $s.f(i)$ est stocké dans R_i au début du calcul. C'est pour cela que l'opération de récursion entre en jeu².

Pour faciliter les instructions $A := R_A$, on va diviser le domaine en trois sous-domaines : $[0, cn^K - 1]$, $[cn^K, 2cn^K - 1]$ et $[2cn^K, 3cn^K - 1]$. On va utiliser le premier intervalle pour stocker $s.f$, le deuxième pour simuler le calcul de M , et le troisième pour extraire la fonction de sortie ; pour ce faire, on va utiliser R_A de manière à encoder la sortie (c'est un usage différent de celui introduit).

On va s'autoriser la définition par cas (on a vu dans un lemme précédent que ça ne rendait pas le LSRS plus puissant) et quelques opérations simples qui ne sont pas directement disponibles dans la définition d'un LSRS. Le système qu'on va proposer n'est pas un LSRS à proprement parler, mais sa traduction en véritable LSRS est immédiate, quoiqu'elle nécessite quelques fonctions en plus.

Pour simplifier la présentation, on va présenter les définitions des fonctions sur les trois intervalles ; elles peuvent être combinées via un LSRS.

Dans un LSRS, l'ordre est important. Ici, les fonctions doivent être présentées dans cet ordre : $I, A, B, N, R_A, I', A', B', N', R'_A$.

Dans l'article d'origine [3] de ce brouillon, les auteurs prennent la convention que l'addition est bornée ; si $a + b > cn$ où cn est la taille du domaine, alors $a + b$ est en fait évalué à 0 dans le LSRS. On peut utiliser cette convention pour simuler une définition par cas sur la taille entière du domaine. Si le LSRS est défini sur $3cn^K$ alors le test $x < cn^K$ peut se simuler avec le test : $x = 0$ ou $x + x + x > 0$.

Ainsi, avant de définir les autres fonctions, intéressons-nous à un moyen d'obtenir cn^K dans le LSRS, ce qui nous sera fort utile pour la suite.

2. C'est l'opération $g, g' \mapsto g' [g^{\leftarrow}(x)]_x$

$$f_0(x) = \begin{cases} 1 & \text{si } x = 0 \\ x + 1 & \text{si } 3x > 0 \\ f_0(x - 1) & \text{sinon} \end{cases} \quad (\text{B.25})$$

La définition de f_0 est telle que $f_0(cn^K - 1) = cn^K$ et pour $x > cn^K$, $f_0(x) = cn^K$.

On peut utiliser cette astuce pour définir, en même temps que la phase d'initialisation, une fonction qui permet d'obtenir cn^K . Si $x < cn^K$, c'est-à-dire, $x = 0$ ou $x + x + x > 0$, on définit A et R'_A par :

$$A(x) = x \quad (\text{B.26})$$

$$R'_A(x) = f(x) \quad (\text{B.27})$$

Les autres fonctions valent 0 sur cet intervalle, sauf $f_0(x) = x + 1$. Pour l'instant, on n'utilise pas $f_0(x)$, qui de toute façon n'est pas prête à l'emploi.

Cette initialisation va permettre de récupérer les valeurs de R_A dans la deuxième partie du domaine.

Sur le domaine $\llbracket cn^K, 2cn^K - 1 \rrbracket$:

$$I(x) = \begin{cases} 1 & \text{si } x = cn^K \\ I'(x - 1) & \text{sinon} \end{cases} \quad (\text{B.28})$$

$$I'(x) = \begin{cases} i & \text{si } I(x) \text{ est de la forme IF}(A = B)\{I(i)\} \text{ ELSE } \{I(j)\} \text{ et } A(x) = B(x) \\ j & \text{si } I(x) \text{ est de la forme IF}(A = B)\{I(i)\} \text{ ELSE } \{I(j)\} \text{ et } A(x) \neq B(x) \\ I(x) & \text{si } I(x) \text{ est de la forme HALT} \\ I(x) + 1 & \text{sinon} \end{cases} \quad (\text{B.29})$$

$$A(x) = \begin{cases} 0 & \text{si } x = cn^K \\ A'(x - 1) & \text{sinon} \end{cases} \quad (\text{B.30})$$

$$A'(x) = \begin{cases} c & \text{si } I(x) \text{ est de la forme } A := c \\ A(x) * B(x) & \text{si } I(x) \text{ est de la forme } A := A + B \\ R_A(x) & \text{si } I(x) \text{ est de la forme } A := R_A \\ N(x) & \text{si } I(x) \text{ est de la forme } A := N \\ A(x) & \text{sinon} \end{cases} \quad (\text{B.31})$$

$$B(x) = \begin{cases} 0 & \text{si } x = cn^K \\ B'(x - 1) & \text{sinon} \end{cases} \quad (\text{B.32})$$

$$B'(x) = \begin{cases} A(x) & \text{si } I(x) \text{ est de la forme } B := A \\ B(x) & \text{sinon} \end{cases} \quad (\text{B.33})$$

$$N(x) = \begin{cases} n & \text{si } x = cn^K \\ N'(x - 1) & \text{sinon} \end{cases} \quad (\text{B.34})$$

$$N'(x) = \begin{cases} A(x) & \text{si } I(x) \text{ est de la forme } N := A \\ N(x) & \text{sinon} \end{cases} \quad (\text{B.35})$$

$$R_A(x) = R'_A[A^{\leftarrow}(x)]_x \quad (\text{B.36})$$

$$R'_A(x) = \begin{cases} B(x) & \text{si } I(x) \text{ est de la forme } R_A := B \\ R_A(x) & \text{sinon} \end{cases} \quad (\text{B.37})$$

Notons que cn^K code le premier instant du calcul.

Les fonctions $I(x)$ et $I'(x)$ ne prennent qu'un nombre fini de valeurs. Ainsi, en écrivant *si* $I(x)$ *est de la forme* $R_A := B$, on écrit en fait *si* $I(x) = i_1$ *ou* $I(x) = i_2$ *... ou* $I(x) = i_m$ où i_1, \dots, i_m sont les numéros des instructions $R_A := B$ dans le programme de la RAM.

De plus, les opérations $I'(x - 1)$ sont en réalité écrites $I[1^{\leftarrow}(x)]_x$ dans un vrai LSRS. Comme on l'a précédemment dit, on utilise R_A pour extraire les valeurs de sortie du calcul de M , donné par la valeur n' de N et le contenu $R(0), \dots, R(n' - 1)$, à la fin du calcul. Pour $2cn^K \leq x < 3cn^K$, le LSRS consiste en les équations suivantes :

$$A(x) = x - 2cn^K \quad (\text{B.38})$$

$$R_A(x) = R'_A[A^{\leftarrow}(x)]_x \quad (\text{B.39})$$

Les autres fonctions du LSRS sont définies par $h(x) = h(x - 1)$ (les fonctions stationnent).

Il ne reste plus qu'à voir que les équations de S définissent les fonctions attendues pour décrire le calcul de M et que S produit la bonne sortie.

Soit s la RAM-structure, $1(-), n(-), id(-)$ ³ comme précédemment, et soient $I, A, B, N, R_A, I', A', B', N', R'_A$ des fonctions qui vérifient les équations de S .

Premièrement, il est clair que pour $a < c(s.n)^K$, on a :

$$A(a) = a \tag{B.40}$$

$$R'_A(a) = s.f(a) \tag{B.41}$$

$$I(a) = B(a) = N(a) = R_A(a) = I'(a) = A'(a) = B'(a) = N'(a) = 0 \tag{B.42}$$

Les valeurs des fonctions au point $c(s.n)^K$ décrivent la configuration de la RAM au début du calcul. De plus, la définition de A et R'_A sur la première partie du domaine permet de rendre compte du fait que M contient $s.f(i)$ dans le registre R_i .

On montre facilement par induction sur t que la valeur des fonctions au temps $c(s.n)^K + t$ sont correctes. La partie la plus difficile réside dans le calcul de R_A . Remarquons premièrement que, puisque $A(a) < c(s.n)^K$ pour chaque a , l'initialisation sur la première partie du domaine assure que $A^{\leftarrow}(a) < a$ pour tout $a \in [cn^K, 2cn^K - 1]$. Deux cas possibles :

- $c(s.n)^K \leq A^{\leftarrow}(a) < a$. Dans ce cas, il existe $b \in [c(s.n)^K, a - 1]$ tel que $A(a) = A(b)$; donc le registre courant R a déjà été visité pendant le calcul, et $A^{\leftarrow}(a)$ est la dernière étape où cela s'est produit ; ainsi $R'_A(A^{\leftarrow}(a))$ donne la bonne valeur de $R_A(a)$.
- $A^{\leftarrow}(a) < c(s.n)^K$. Dans ce cas, il n'y a pas de $b \in [c(s.n)^K, a - 1]$ tel que $A(a) = A(b)$; donc le registre courant R n'a pas été visité pour le moment, et devrait toujours contenir la valeur de $f(A^{\leftarrow}(a))$ ⁴. Par initialisation, il découle que $A^{\leftarrow}(a) = A(a)$ et $R'_A(A(a)) = f(A(a))$, comme voulu.

Enfin, on doit encore montrer que S définit correctement $\Gamma(s)$. Par définition de R_A et N sur la troisième partie du domaine du LSRS, pour chaque a , $R_A(2c(s.n)^K + a)$ contient les valeurs de du registre R_a à la fin du calcul. De plus, $N(3c(s.n)^K - 1)$ contient la valeur de N à la fin du calcul. Ainsi, avec une projection bien choisie, $\Gamma(s)$ peut être extraite des fonctions définies par S .

Ainsi, on a montré que S calcule correctement $\Gamma(s)$. Donc Γ est n^K -représentée par S .

□

3. On obtient la fonction $i \mapsto cn^K$ avec une astuce. En fait, le n^K doit être explicité, et on peut le faire, même si cette preuve le passe sous silence pour des raisons de clarté. On peut calculer cn^K en rajoutant quelques équations : c et K étant fixés, si on a la multiplication, il suffit de rajouter K équations $f_1(x) = n(x)$ et $f_{i+1}(x) = f_i(x) \times n(x)$, puis $c - 1$ équations $f_{K+j+1}(x) = f_{K+j}(x) + f_K(x)(x)$; on obtient alors cn^K dès le premier tour de calcul. En revanche, si on n'a que l'addition, on ne peut pas reproduire ce schéma. On peut envisager d'étudier g non pas sur $[0, 3cn^K - 1]$ mais sur $[0, 4cn^K - 1]$, et de couper ce domaine en quatre ; le premier domaine servirait alors à calculer cn^K (mais comment délimiter ce premier domaine explicitement, avec un LSRS ?) et le reste serait le même que présenté ici.

4. Petite erreur de frappe dans l'article original ? Il y est écrit : $f(A(a))$.

Bibliographie

- [1] Stephen Bellantoni and Stephen Cook. A new recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2(2) :97–110, 1992.
- [2] Alan Cobham. The intrinsic computational difficulty of functions. *Logic, methodology and philosophy of science, Proceedings of the 1964 International Congress. Studies in logic and the foundations of mathematics*, page 24–30, 1965.
- [3] 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.
- [4] Etienne Grandjean. Invariance properties of rams and linear time. *Computational Complexity*, 4(1) :62–106, 1994.
- [5] Kenneth Kunen. Set theory : an introduction to independence proofs, 1980.
- [6] 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.
- [7] Thomas Schwentick. Algebraic and logical characterizations of deterministic linear time classes. In *Proceedings of the 14th Annual Symposium on Theoretical Aspects of Computer Science*, STACS '97, pages 463–474, London, UK, UK, 1997. Springer-Verlag.
- [8] Peter van Emde Boas. Handbook of theoretical computer science (vol. a). chapter Machine Models and Simulations, pages 1–66. MIT Press, Cambridge, MA, USA, 1990.