M2 LFMI

MEMOIRE DE STAGE

Unification des modèles de calculs caractérisant le temps polynomial

BEURIER Erwan

Année 2015-2016

Disclaimer

Ceci n'est qu'un brouillon de mon mémoire. Il a beau dépasser les 40 pages, il n'en est pas moins incomplet. Il est probablement plein de fautes de français et de textes en rouge voire encadrés en rouge. C'est moi-même qui me signale des points que je ne dois pas oublier de rectifier ou d'ajouter.

Je n'aime pas que les liens hypertextes soient trop visibles, mais je n'ai pas encore cherché comment on pouvait les rendre visibles ET discrets, donc pour l'instant, 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.

Notations

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 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

Chapitre 1

Unification de deux caractérisations de P

1.1 Des caractérisations de P

1.1.1 Notions générales

- Algèbre
- Fonctions récursives sur une algèbre

1.1.2 Approche de Bellantoni et Cook

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

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

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

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

et étant close par schéma de récursion safe
Décaler d'une tabulation
f est définie par récurrence *safe* à partir de $\left(g_c\right)_{c\in\sigma}\in$ BC lorsque pour tout constructeur c :

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

et composition safe.
Décaler d'une tabulation
f est définie par composition *safe* à partir de $h,g_0,g_1\in$ BC lorsque :

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

Théorème 2 (Bellantoni et Cook). *[1] BC = P*

Parler de la preuve.

1.1.3 Approche de Cobham

Toujours dans le cas des mots : $\sigma=\{C_1,\ldots,C_k\}$ où $\min\limits_{i\in k}\left(C_i\right)=0$ et $\max\limits_{i\in k}\left(C_i\right)=1$.

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

Décaler d'une tabulation

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

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

THÉORÈME 4 (Cobham). *Cob = P*

Parler de la preuve.

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

1.1.4 Approche de Leivant

Résumé de l'article [4] de Leivant

.

Définition 5 (Algèbre [4])**.** *Soit* $\sigma = \{C_1, \ldots, C_k\}$ *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* $\tau = C_i(\tau_1, \ldots, \tau_{r_i})$ *alors* C_i *est le constructeur extérieur de* τ*.*

Exemples.

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

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

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

Une fonction est définie par récurrence ramifiée lorsque :

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

On note $\text{TRec}_2(\mathbb{A})$ *le sous-ensemble de* $\text{TRec}(\mathbb{A})$ *dont chaque fonction est constructible en n'utilisant que deux tiers* $\mathbb{A}_0, \mathbb{A}_1$*.*

Un résultat intéressant est que si une fonction $f \in \text{TRec}(\mathbb{A})$ a des arguments de tiers inférieur à son tier d'arrivée, alors f *ignore* ces arguments.

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

Définition 7 (Degré d'imbrication de récurrences)**.** *Soit* $f \in \text{TRec}(\mathbb{A})$ [1]*.*

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 :*

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

- *Si tier(h) < tier(g) alors* $\delta(f) = \delta(g)$;
- *Si tier(h) = tier(g) alors* $\delta(f) = \max(\delta(g), \delta(h))$;
- *Si tier(h) > tier(g) alors* $\delta(f) = \max(1, \delta(h)) \times \delta(g)$;

- *Si f est définie par récurrence ramifiée* $f(c_i^j(a_1, \ldots, a_{r_i}), \bar{x}) = g_{c_i}(f(a_1, \bar{x}), \ldots, f(a_{r_i}, \bar{x}), \bar{a}, \bar{x})$, *telles que* $\left(g_{\alpha_j}\right)_{j \in p}$ *aient des arguments critiques et* $\left(g_{\beta_j}\right)_{j \in q}$ *n'en aient pas, alors* $\delta(f) = \max\left(1 + \delta\left(g_{\alpha_1}\right), \ldots, 1 + \delta\left(g_{\alpha_p}\right), \delta\left(g_{\beta_1}\right), \ldots, \delta\left(g_{\beta_q}\right)\right)$.

Il définit aussi une RAM personnalisée pour la construction de termes, voir la définition 13. Cette RAM, notée $\mathbb{A}$-RAM dans ce mémoire, coïncide avec les autres modèles de calcul sur la classe P, mais son manque de test d'égalité et sa mémoire limitée la rendent moins puissante qu'une RAM au sens de Grandjean [2] 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 [4] qui nous intéressent dans le cadre de ce mémoire :

THÉORÈME 8. *Soit $\mathbb{A}$ une algèbre de mots. Soit $f : \mathcal{A} \to \mathbb{A}$.*
Les propositions suivantes sont équivalentes :

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

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

La récurrence ramifiée engendre P et deux niveaux de ramification suffisent.
Le théorème 8 a été prouvé pour une $\mathbb{A}$-RAM ; l'un des objets de ce stage a aussi été d'obtenir un résultat similaire pour la σ-RAM, voir les chapitres 2 et 3.

1.1.5 Correspondance entre Bellantoni et Cook et Leivant

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

LEMME 10. *Soit $f \in P$, f fonction sur $\mathbb{W}$ (mots binaires).*
Un argument x de f est normal *dans BC si et seulement si tier$(x) >$ tier(f) dans $TRec(\mathbb{W})$.*
Un argument x de f est safe *dans BC si et seulement si tier$(x) \leq$ 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* dest $(; C(; x))$ est définissable par récurrence plate (sans appel récursif) dans TRec $(\mathbb{W})$:

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

Une ramification triviale donne dest : $\mathbb{W}_i \to \mathbb{W}_i$.
De même, la conditionnelle *safe* if $(; x, y, z)$ est définissable par récurrence plate dans TRec $(\mathbb{W})$:

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

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

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

Remarquons d'abord que, si une fonction $f_{i,j} : \mathbb{W}_i \to \mathbb{W}_j$ est définissable par récurrence ramifiée en utilisant des tiers dans $[\![0, k]\!]$, alors il existe $f_{i+h,j+h} : \mathbb{W}_{i+h} \to \mathbb{W}_{j+h}$ 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 $f_{i+h,j} : \mathbb{W}_{i+h} \to \mathbb{W}_j$ pour tout $h \in \omega$ (il suffit pour cela d'utiliser une fonction dite de coercion $\kappa_{i,j} : \mathbb{W}_i \to \mathbb{W}_j, i > j$, 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 : $f : \overline{\mathbb{W}_k} \times \overline{\mathbb{W}_j} \to \mathbb{W}_j$.

Pour la définition par récurrence, le raisonnement est similaire.

Passage de TRec (𝕎) à BC

Lemme 11 (Leivant [4]). *Soit $f : \mathbb{A}_i \times \mathcal{A} \to \mathbb{A}_j$. Si $i < j$ alors $\forall x, f(x, \bar{x}) = f(C, \bar{x})$ où C est un constructeur d'arité 0 de $\mathbb{A}$.*

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

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

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

Le cas des constructeurs et des projections est trivial.

Si $f(\underset{1}{\bar{x}}, \underset{0}{\bar{y}}) = g(\underset{1}{\bar{x}}, \underset{1}{h_1}(\bar{x}), \underset{0}{\bar{y}}, \underset{0}{h_0}(\bar{x}, \bar{y}))$ alors :

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

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

L'équivalence entre les deux se généralise sans problème aux algèbres de mots voire aux algèbres quelconques (bien que dans le cas d'algèbres quelconques, on ne sache pas s'il y a équivalence entre P et BC ou TRec (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 12 (σ-RAM [2])**.** *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 a priori *des valeurs entières.*
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* $op \in \sigma$
- $A := N$
- $N := A$
- $A := R_A$
- $B := A$
- $R_A := B$
- $IF(A = B)\{I(i)\}$ $ELSE$ $\{I(j)\}$
- $HALT$

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 permet d'effectuer plusieurs instructions en même temps.

Entrées et sorties. Dans leur article [2], Grandjean et Schwentick indiquent que leur σ-RAM prend en entrée une structure $(\llbracket 1, n\rrbracket, f_1, \dots, f_k, C_1, \dots, C_l)$ 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 $i \in \llbracket 1, k\rrbracket$ et $j \in \llbracket 1, n\rrbracket$
- $R_{k\times n+j} := C_i$ où $i \in \llbracket 1, l\rrbracket$

(La taille est stockée dans N et les fonctions sont stockées dans l'ordre de numérotation).
La sortie est une nouvelle structure $(\llbracket 1, n'\rrbracket, f'_1, \dots, f'_{k'}, C'_1, \dots, C'_{l'})$ stockée de manière similaire dans la σ-RAM.

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

Soit $\mathbb{A}$ une σ-algèbre.

Définition 13 ($\mathbb{A}$-RAM [4])**.** *Une $\mathbb{A}$-RAM est un modèle de calcul comprenant :*

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

Les registres contiennent des termes de l'algèbre $\mathbb{A}$. Par défaut, on leur assigne une valeur d parmi les constructeurs d'arité 0.
Un programme de $\mathbb{A}$-RAM est un ensemble fini d'instructions dont chacune est de l'une des formes suivantes :

- *(const)* $s_a \pi_{j_1} \dots \pi_{j_{r_i}} C_i \pi_j s_b$
- *(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, selon le résultat, place la $\mathbb{A}$-RAM dans l'état s_{b_i} si le constructeur extérieur de $*j$ est C_i.

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

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

1.2.3 Comparaison entre les deux RAM

Modèles Turing-complets

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

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

Les démonstrations se trouvent dans [4] Il me faut une source pour la σ-RAM..

$\mathbb{A}$-RAM dans σ-RAM

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

Premières observations. Premièrement, la σ-RAM calcule des entiers alors que la $\mathbb{A}$-RAM construit des termes. Cette remarque plutôt intuitive est pourtant la cause *meta* des différences de puissance de ces machines. La 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 frustre car il l'a construite pour qu'elle colle parfaitement à sa caractérisation de $DTIME_{\mathbb{A}}(n^k)$.

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. Troisièmement, la σ-RAM dispose d'un test d'égalité gratuit, ce qui n'est pas le cas de la $\mathbb{A}$-RAM. Enfin, la $\mathbb{A}$-RAM a une commande de destruction de terme qui n'a pas réellement d'équivalent dans la 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 15 (σ-RAM constructrice)**.** *Une σ-RAM constructrice est un modèle de calcul composé de :*

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

Les instructions sont similaires à celles de la 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*$(A_1 = A_2)\{I(i)\}$ *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 les deux. On suppose alors que R_{A_1} est le registre dont l'indice est l'image du terme contenu dans A_1 par la bijection. Sinon, on peut considérer que les registres sont numérotés directement par des termes.

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

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

LEMME 16. *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 GOTO$(I(j_1), \ldots, I(j_t))$ 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) $s_a \pi_{j_1} \ldots \pi_{j_{r_i}} C_i \pi_j s_b$

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

Algorithm 1: Simulation de la commande (const)

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

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

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

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

Cette simulation se fait en 6 étapes de calcul.

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

```
// On détruit entièrement le terme et on récupère chacun des arguments de son constructeur
   extérieur de R_{f(j)}. Si l'arité du constructeur est strictement inférieure à p, on assigne une
   valeur par défaut au registre.
for p from r to 1 do
    A_1 := f(j);
    A_1 := R_{A_1};
    A_p := dest_p(A_1);
end
for i from 1 to k do
    // On construit un terme :
    A_1 := C_i(A_1, ..., A_{r_i});
    // On le met de côté, à disposition, quitte à écraser le contenu de A_2 :
    A_2 := A_1;
    // On récupère le terme qu'on voulait analyser :
    A_1 := f(j);
    A_1 := R_{A_1};
    if A_1 = A_2 then
        // Alors on a reconstruit le même terme ; ça veut dire qu'on est au bon i. Dans ce cas, on
           va à l'instruction j_i :
        IF(A_1 = A_2){I(j_i)} ELSE {I(j_i)};
        // Astuce pour avoir un GOTO
    else
        // Sinon, on s'est trompé. Dans ce cas, on récupère l'argument qu'on venait d'effacer dans
           A_2 :
        A_1 := dest_2(A_1);
        A_2 := A_1;
        // On récupère aussi le premier sous-terme, pour pouvoir faire une nouvelle construction
           par la suite :
        A_1 := f(j);
        A_1 := R_{A_1};
        // Et on continue la boucle.
    end
end
// On construit un nouveau terme en regardant chaque constructeur et on vérifie s'il est égal au
   terme de R_{f(j)}
```

Algorithm 3: Simulation de la commande (switch)

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

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

σ-RAM dans $\mathbb{A}$-RAM

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

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

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

Input: $\alpha, \beta, \pi_1, \ldots, \pi_r$
// Détruit le terme contenu dans α, stocke ses composantes dans chaque π_p, puis reconstruit le terme dans β.
for p *from* 1 *to* r **do**
  Récupérer le p-ième sous-terme de α et le stocker dans π_p ;
end
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 $\pi_1, \ldots, \pi_r$.

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

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

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

Input: $\alpha, \beta, s_1, s_0, \pi_1, \pi_2, \pi'_1, \pi'_2$
/* 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. */
/* */
/* Vérifie si $\alpha = \beta$ en faisant une analyse inductive sur la construction des termes qu'ils contiennent. */

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.
while π_1 *et* π_2 *sont non vides* **do**
  if π'_1 *et* π'_2 *ont les mêmes constructeurs extérieurs* **then**
    if π'_1 *et* π'_2 *ont des sous-termes* **then**
      Empiler les sous-termes de π'_1 dans π_1 ;
      Empiler les sous-termes de π'_2 dans π_2 ;
    end
    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 ;
  else
    Aller à l'état s_1 ;
  end
end
Aller à l'état s_0 ;

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

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

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

$\mathrm{MEM}(f(i_1), R_{f(i_1)}, \mathrm{MEM}(\ldots, \mathrm{MEM}(f(i_n), R_{f(i_n)}, \varepsilon)\ldots))$

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

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

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

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. 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.

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

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

// Initialisation de la mémoire
Copier μ dans μ_3 ;
Vider μ_4 ;
Stocker le premier indice et la première valeur de μ dans μ_1 et μ_2 ;

// Première boucle : on avance dans la mémoire en vérifiant à chaque fois si on est au bon indice.
while μ_3 *est non-vide* **do**
    Empiler (μ_1, μ_2) dans μ_4 ;
    Stocker le premier indice de μ_3 dans μ_1 ;
    Stocker la première valeur de μ_3 dans μ_2 ;
    Dépiler μ_3 ;
    if $\alpha = \mu_1$ **then**
        Sortir de la boucle While ;
    end
end
Empiler (μ_1, β) dans μ_3 ;

// Deuxième boucle : étape inverse : on récupère tout ce qu'on a visité précédemment.
while μ_4 *est non-vide* **do**
    Empiler (μ_1, μ_2) dans μ_3 ;
    Stocker le premier indice de μ_4 dans μ_1 ;
    Stocker la première valeur de μ_4 dans μ_2 ;
    Dépiler μ_4 ;
end
Copier μ_3 dans μ ;

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

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

Input: $\mu, \mu_1, \mu_2, \mu_3, \mu_4, \alpha, \beta, \pi_1, \pi_2, \pi'_1, \pi'_2$
/* $\mu, (\mu_i)_{i\in 4}$ sont tels que décrits plus haut. */
/* α est l'indice auquel on veut insérer. */
/* β est la valeur qu'on veut insérer à l'indice α. */
/* $\pi_1, \pi_2, \pi'_1, \pi'_2$ sont les registres de travail du test d'égalité. */
// Initialisation de la mémoire
Copier μ dans μ_3 ;
Vider μ_4 ;
Stocker le premier indice et la première valeur de μ dans μ_1 et μ_2 ;

// Boucle : avancer dans la mémoire jusqu'à trouver le bon indice.
while μ_3 *est non-vide* **do**
  Stocker le premier indice de μ_3 dans μ_1 ;
  Stocker la première valeur de μ_3 dans μ_2 ;
  Dépiler μ_3 ;
  if $\alpha = \mu_1$ **then**
    Sortir de la boucle While ;
  end
end
Copier μ_2 dans β ;

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

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

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 $(\alpha_i)_{i\in k}$, un registre ν, cinq registres spéciaux dédiés à la gestion de la mémoire $\mu, (\mu_i)_{i\in 4}$, $r+4$ registres de travail $\overline{\pi''}, \pi_1, \pi_2, \pi'_1, \pi'_2$.

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

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

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

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

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

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

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

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

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 $\mathbb{A}$-RAM doit utiliser des astuces pour reproduire cet accès du mieux possible.

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

Code + algo de cette optimisation

Une tentative d'amélioration

Enfin, on peut penser à donner un test d'égalité gratuit à la $\mathbb{A}$-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 $\mathcal{O}\left(|\mu|\right)$;
- $R_{A_1} := A_i$ se simule à présent en temps $\mathcal{O}\left(|\mu|\right)$;
- $\text{IF}(A_1 = A_2)\{I(i)\}$ ELSE $\{I(j)\}$ se simule à présent en temps $\mathcal{O}\left(1\right)$;
- Les autres instructions ne sont pas affectées.

De plus, le test d'égalité gratuit de la $\mathbb{N}$-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 $\mathbb{A}$-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 $\text{MEM}\left(-, \ldots, -\right)$, où chaque emplacement de la mémoire correspond à un constructeur). L'adresse est alors un mot et se traduit comme un ensemble de destructions successives du terme de mémoire, en prenant garde à récupérer le sous-terme correspondant au constructeur détruit.

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

Problème majeur

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

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

Chapitre 2

Vers une caractérisation algébrique du temps polynomial sur σ-RAM

2.1 Préliminaires

(p.198)

2.1.1 Structures de RAM

(p.198)

Définition 18 (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 $C \in t$;*
- $f : n \to \mathbb{N}$ *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 $i \in n$.

Définition 19 (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-bornées de type t_2 [1].

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

2.1.2 Machine RAM

(p.200)

La machine RAM reste la même (heureusement). On va utiliser la $\{+\}$-RAM ou des versions un brin plus puissantes comme la $\{+, -, \times, \div k\}$-RAM pour un $k \in \mathbb{N}$ fixé [2].

Définition 20 (Temps polynomial)**.** *On définit $DTIME_{RAM}\left(n^H\right)$ comme étant l'ensemble des fonctions calculables sur $\{+\}$-RAM en temps $\mathcal{O}\left(n^K\right)$, telles que le nombre de registres utilisés, les valeurs entières manipulées (y compris les adresses de registres) soient bornés par $\mathcal{O}\left(n^K\right)$.*

2.1.3 Réductions affines

(p.202)

On laisse inchangées les notions de *transformations affines* (définition 2.3), de *réductions affines* (définition 2.5), de *projections affines* (définition 2.7). Les théorèmes et lemmes suivants ou intermédiaires sont aussi inchangés. Ils permettront de définir des réductions qui *restent* dans $\mathrm{DTIME}_{\mathrm{RAM}}\left(n^H\right)$ pour un k fixé.

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

2. L'ajout de ces opérations ne rend pas la $\{+\}$-RAM plus puissante [3], [2].

Définition 21 (Fonction affine non-décroissante)**.** *On appellera* fonction affine non-décroissante*, ou plus simplement,* fonction affine*, une fonction A de la forme $A(x_1, \ldots, x_k) = a_0 + a_1x_1 + \cdots + a_kx_k$, telle que soit $a_1, \ldots, a_k \geqslant 0$ et $a_0 \in \mathbb{Z}$, soit $a_1, \ldots, a_k = 0$ et $a_0 \geqslant 0$.*

Définition 22 (Transformation affine)**.** *Soit T une fonction de RAM qui envoie des RAM-structures de type t_1 sur des RAM structures de type t_2.*

On dit que T est une transformation affine lorsque :
REMPLIR ICI!!!!!!!!!!!!

-

2.2 Le framework algébrique

(p.208)

2.2.1 LSRS

(p.208)
Le LSRS en tant que tel n'a pas l'air collé à la définition du temps linéaire. On garde la définition pour le moment.

Définition 23 (Application bornée et *equal-predecesor*)**.** *Pour $f : n \to \mathbb{N}$, on définit deux opérations :*

- *L'application bornée :*
$$f[x]_y = \begin{cases} f(x) & si\ x < y \\ x & sinon \end{cases}$$
- *L'opération* equal-predecesor *:*
$$f^{\leftarrow}(x) = \begin{cases} \max\left(\{y < x | f(x) = f(y)\}\right) & si\ un\ tel\ y\ existe \\ x & sinon \end{cases}$$

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

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

$f(x) = g'(y)$ où y est de plus grand z tel que $g(x) = g(z)$, ou $f(x) = x$ si un tel y n'existe pas.

Définition 24 (LSRS)**.** *Soit F un ensemble de symboles de fonctions (dites* fonctions de base*), soient $f_1, \ldots, f_k$ des symboles de fonctions qui n'apparaissent pas dans F. Pour $i \leqslant k$, notons $F_i = F \cup \{f_1, \ldots, f_i\}$.*

*Un LSRS (*Linear Simultaneous Recursion Scheme*) S sur $f_1, \ldots, f_k$ 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 $* \in \{+, -, \times\}$* [3]
- *(récursion) $f_i(x) = g'\left[g^{\leftarrow}(x)\right]_x$ où $g' \in F_k$ et $g \in F_{i-1}$*

On doit refaire la définition 3.3 de l'article (la définition de *linéairement représentable*).
Pour t un type, on note F_t l'ensemble des symboles de fonctions suivants : $\{1(-), n(-), id(-)\} \cup \{f_C | C \in t\} \cup \{f | f \in t\}$.

Remarque 1. Soient t un type et S un LSRS pour $f_1, \ldots, f_k$ 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 $\{f_1, \ldots, f_k\}$.

Ici, on a plusieurs possibilités pour adapter le concept de *linéairement représenté* (définition 3.3) à n^K.

3. Les opérations peuvent varier, il est dit dans [2] que l'on peut choisir n'importe quelle opération binaire calculable en temps linéaire sur machine de Turing, voire la multiplication. Nous verrons d'ailleurs par la suite que, pour des raisons de facilité, nous aurons besoin de la multiplication.

Définition 25 (RAM n^K-représentée par LSRS - Proposition 1 [4])**.** *Soient t_1, t_2 des types. Soit Γ une (t_1, t_2)-fonction de RAM.*

Soit S un LSRS pour $f_1, \ldots, f_k$ 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, S définit des fonctions $f_1, \ldots, f_k : c(s.n)^K \to c(s.n)^K$ telles que $\Gamma(s) = P((s.n)^K, S(s))$ ($S(s)$ est la structure définie par le LSRS).

La modification réside dans le domaine de définition des fonctions et la taille de la sortie dans la projection.

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

La définition 3.4 (définition par cas) reste la même et les lemmes 3.5 (la définition par cas ne change pas la puissance des LSRS) et 3.6 (composition de fonctions définissables par LSRS reste définissable par LSRS) restent valables.

2.2.2 LRS

(p.211)

La définition d'un terme récursif (non numérotée) et d'un LRS (définition 3.7) restent les mêmes. Le lemme 3.8 d'existence d'une solution unique au LRS aussi.

On doit adapter la définition de n^K-représentée par LRS :

Définition 26 (RAM n^K-représentée par LRS)**.** *Soit Γ une fonction de RAM.*

Soit E un LSRS $g(x) = \sigma(x)$.

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 $g : c(s.n)^K \to c(s.n)^K$ telle que $\Gamma(s) = P((s.n)^K, E(s)$ ($E(s)$ est la structure définie par le LRS, elle est de type $\{g\}$).

La modification réside dans le domaine de définition de g.

Et là, c'est la foire.

On doit vérifier si le théorème principal de l'article tient encore au temps polynomial.

Conjecture 1 (Adaptation du lemme 3.10 (p.212)). *Toute fonction de RAM n^K-représentée par LSRS est aussi n^K-représentable par LRS.*

Démonstration. Ici commence la relecture de la preuve.

Soit Γ une fonction de RAM n^K-représentée par un LSRS S sur F_{t_1} pour $f_0, \ldots, f_{k-1}$, 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 $s'' = \Gamma(s) = P\left((s.n)^K, s'\right)$. Pour simplifier les notations, on va simplement écrire n au lieu de $s.n$.

L'idée est de coder les fonctions $f_0, \ldots, f_{k-1} : cn^K \to cn^K$ 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 disjointes. 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, *by function values of a contiguous interval* (???).

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

- pour tout $b \in kcn^K$, on a $g(b) = b \div k$;
- pour tous $a \in cn^K$ et $i \in k$, on a $g\left(kcn^K + ka + i\right) = kcn^K + kf_i(a) + i$ (⋆)

4. Il pourra à l'avenir y avoir d'autres possibilités mais cette définition semble bien élargir la définition pour le temps linéaire.

On a donc besoin d'un moyen de calculer n^K dans la fonction de sortie.
Pour l'instant, pour des raisons de simplicité, puisqu'on sait que la fonction est n^K-représentable, on va supposer qu'on a accès à une fonction $i \mapsto n^K$. Pour ce faire, on peut soit la définir par multiplication en rajoutant K équations *(ce qui ne me semble pas naturel, parce qu'on modifie le LSRS à chaque fois qu'on veut changer la taille d'arrivée ?)* soit on peut rajouter une fonction constante $i \mapsto n^K$ *(on ajoute ou remplace une fonction dans les fonctions de base du LSRS ? Ça veut dire qu'on rajoute un symbole de fonction au type de départ, qui sait déjà ce qu'on va en faire ?)*.
Problème : apparemment le LSRS sait qu'il représente des fonctions de taille n^K, mais est-ce suffisant pour justifier l'insertion de cette fonction en tant que fonction du domaine ? Apparemment, un même LSRS peut être utilisé pour toutes les tailles de domaines, donc peut-être qu'on peut, au cas par cas, rajouter une fonction en plus selon le domaine d'arrivée ? (La question ici est de savoir si on peut rajouter naturellement cette fonction constante d'accès direct au max, sans avoir l'air de tricher pour se faciliter la vie.)
Ou bien on peut voir l'ajout de cette fonction comme un ajout de contrainte. C'est l'ajout de cette fonction qui impose que le domaine d'arrivée sera de taille $\mathcal{O}\left(n^K\right)$.
Solution Dans la démonstration originale, kcn est calculé à la volée dans le LRS. Il s'agit d'une abréviation : $cn = \underbrace{n(y) + \cdots + n(y)}_{c \text{ fois}}$, et $kcn = \underbrace{cn + \cdots + cn}_{k \text{ fois}}$.
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, on ne peut pas faire un terme similaire pour n^K, car on ne peut pas autoriser un terme dont la longueur dépend de l'entrée (un LRS est défini par une équation fixée dès le départ). Il faut donc qu'on ajoute la multiplication comme opération de base de la définition par LRS, et on pourra écrire un terme similaire : $n^K = \underbrace{n(y) \times \cdots \times n(y)}_{K \text{ fois}}$, puis $cn^K = \underbrace{n^K + \cdots + n^K}_{c \text{ fois}}$, et $kcn^K = \underbrace{cn^K + \cdots + cn^K}_{k \text{ fois}}$.
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 [2].

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

1. Renumérotons les symboles de $F_{t_1} = \{f | f \in t_1\} \cup \{f_C | C \in t_1\} \cup \{1(-), n(-), id(-)\}$ ainsi $\{f^0, \ldots, f^{l-1}\}$. On va limiter les occurrences de ces symboles. Premièrement, les symboles $f_0, \ldots, f_{k-1}$ sont remplacés par $f^l, \ldots, f^{l+k-1}$ respectivement. Ensuite, on va introduire l nouvelles fonctions $f_0, \ldots, f_{l-1}$ 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 $y \mod k$.

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

   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 $y \mod k$ pour $kcn^K \leqslant y < 2kcn^K$, puisque $(y - kcn^K) - kg(y - kcn^K) = (y - kcn^K) - \sum_{j=1}^{k} g(y - kcn^K)$. (Et alors ? ? ?)

   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 $\sigma_i(y) = kcn^K + kg\left(y - kcn^K\right) + i$;

– Si le terme de droite de l'équation est $f^i(x)$, alors $\sigma_i(y) = kcn^K + kf^i\left(g\left(y - kcn^K\right)\right) + i$.
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 $g(b - kcn) = g(ka + i) = (ka + i) \div k = a$, 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 :

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

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 $a \in cn^K$, $b = kcn + ka + i$, 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 :

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

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 $\tau(y) = \left(g\left(b - \delta\right) - kcn^K - j\right) - \left(g\left(b - \delta'\right) - kcn^K - j'\right)$ 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 $f_i(x) = f_{j'}\left[f_j^{\leftarrow}(x)\right]_x$, où $j < i$ et on suppose sans perte de généralité que $i \leqslant j'$ [5] , alors $\sigma_i(y)$ est définie par :

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

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-Predecesor*. 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 $b - \delta = kcn^K + ka + i - (i - j) \underset{i>j}{=} kcn^K + ka + j$. 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 $g\left(kcn^K + ka' + j\right) = g\left(kcn^K + ka + j\right)$. La définition de g assure que, pour chaque $e \geqslant kcn^K$, on a $(g(e) \mod k = j \Leftrightarrow e \mod k = j)$ et $\forall e, e' \in 2kcn^K$, si $g(e) = g(e')$, alors soit $e, e' < kcn^K$, soit $e, e' \geqslant kcn^K$. Donc $g^{\leftarrow}\left(kcn^K + ka + j\right) = kcn^K + ka + j$ et $g^{\leftarrow}(b - \delta) + \delta' = kcn^K + ka + j' \geqslant kcn^K + ka + i = b$. Ainsi :

$$\begin{aligned} \sigma_i(b) &= g\left[g^{\leftarrow}(b - \delta) + \delta'\right]_b - j' + i && (2.1)\\ &= \left(kcn^K + ka + j'\right) - j' + i && (2.2)\\ &= kcn^K + ka + i && (2.3)\\ &= kcn^K + kf_{j'}\left[f_j^{\leftarrow}(a)\right]_a + i && (2.4)\\ &= kcn^K + kf_i(a) + i && (2.5)\\ &= g(b), && (2.6) \end{aligned}$$

comme voulu.

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

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

Donc :

5. 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 $f_i(x) = f_l\left[f_j^{\leftarrow}(x)\right]_x$

$$\begin{aligned}
\sigma_i(b) &= g\left[g^{\leftarrow}(b-\delta)+\delta'\right]_b - j' + i && (2.1)\\
&= g\left(kcn^K + ka' + j'\right) - j' + i && (2.2)\\
&= \left(kcn^K + kf_{j'}(a') + j'\right) - j' + i && (2.3)\\
&= kcn^K + kf_{j'}(a') + j' + i && (2.4)\\
&= kcn^K + kf_{j'}\left[f_j^{\leftarrow}(a)\right]_a + i && (2.5)\\
&= kcn^K + kf_i(a) + i && (2.6)\\
&= g(b), && (2.7)
\end{aligned}$$

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 $\Gamma(s).n = P\left((s.n)^K, s'\right).n = s'.f_j\left(\alpha\left((s.n)^K\right)\right)$, et soient $i < k$ et A une fonction affine tels que, pour toute structure s et tout $a < \Gamma(s).n$, on ait $\Gamma(s).h(a) = P\left((s.n)^K, s'\right).h(a) = s'.f_i\left(A\left((s.n)^K, a\right)\right)$ [6].

   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. 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 :

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

   Il découle de l'équation (⋆) (la définition de g sur kcn^K) et de cette définition que, pour tout $a < cn^K$, on a [7] :

$$\begin{aligned}
g\left(2kcn^K + a\right) &= g\left[g\left[k\left(\left(2kcn^K + a\right) - 2kcn^K\right) + kcn^K + i\right]_{2kcn^K+a} - kcn^K\right]_{2kcn^K+a} && (2.1)\\
&= g\left[g\left[ka + kcn^K + i\right]_{2kcn^K+a} - kcn^K\right]_{2kcn^K+a} && (2.2)\\
&= g\left[g\left(ka + kcn^K + i\right) - kcn^K\right]_{2kcn^K+a} && (2.3)\\
&= g\left[\left(kcn^K + kf_i(a) + i\right) - kcn^K\right]_{2kcn^K+a} && (2.4)\\
&= g\left[kf_i(a) + i\right]_{2kcn^K+a} && (2.5)\\
&= f_i(a) && (2.6)
\end{aligned}$$

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

   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 :

$$\Gamma(s).n = s'''.g\left(\alpha\left(cn^K\right) + (2k+1)cn^K\right)$$ [8]

6. Ça veut juste dire : on donne 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. Reste à voir si cette définition a bien du sens, mais si ça a du sens, alors il n'y a pas de problème ici.

7. Analysons :

- (1) → (2) : parce que soustraction propre.
- (2) → (3) : par définition de l'opérateur *application bornée*.
- (3) → (4) : parce que $ka + kcn^K + i < 2kcn^K$ donc on reprend la définition de la fonction g sur l'intervalle $2kcn^K$.
- (4) → (5) : *because* soustraction propre.
- (5) → (6) : parce que $f_i(a) < cn^K$ et on a le bon décalage avec i.

8. C'est bien ce qu'il nous faut, parce que $\alpha' : a \mapsto \alpha(ca) + (2k+1)ca$ est bien une fonction affine telle que $P'\left(n^K, s'\right).n = s'''.g\left(\alpha'\left(n^K\right)\right)$.

et [9] [10] :

$$\begin{aligned}\Gamma(s).h(a) &= P\left(n^K, s'\right).h(a) && (2.1)\\ &= s'.f_i\left(A\left(n^K, a\right)\right) && (2.2)\\ &= s'''.g\left(2kcn^K + A\left(n^K, a\right)\right) \text{ [11]} && (2.3)\end{aligned}$$

□

2.2.3 Bilan

La démonstration a l'air de marcher ! !

9. On rappelle que s' est la structure résultante du LSRS originel, de type $\{f_0, \ldots, f_{k-1}\}$, qu'on a compactée dans une seule structure de type $\{g\}$.
10. Analysons :
    - (1) → (2) : par définition de A et i.
    - (2) → (3) : parce qu'on a $f_i(a) = g\left(2kcn^K + a\right)$ par construction de g.
11. C'est bien ce qu'il nous faut parce que $A' : a, b \mapsto 2kcb + A(b, a)$ est bien une fonction affine telle que $P'\left(n^K, s'\right).h(a) = s'''.g(A'(n^K, a))$.

2.3 LRS et temps polynomial

(p.216)

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

1. $\Gamma \in DTIME_{RAM}\left(n^K\right)$.
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 avant. La preuve de $3 \Rightarrow 1$ se trouve ci-dessous. La preuve de $1 \Rightarrow 2$ se trouve plus loin.

2.3.1 n^K-représentable par LRS $\Rightarrow$ calculable en temps $\mathcal{O}\left(n^K\right)$

Conjecture 3. *Si une fonction de RAM Γ est n^K-représentable par LRS alors $\Gamma \in DTIME_{RAM}\left(n^K\right)$.*

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 une terme de récursion, et qu'il existe une constante c (et H ?) telles que la sortie $s' = \Gamma(s)$ peut être extraite de $(s.n)^K$ [12]et $g : cn^K \to cn^K$.

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 [13].

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 $c(s.n)^K$ [14]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 $j < p$, $G[j] = \begin{cases} g(j) & \text{si } j \leqslant i \\ j & \text{si } i < j < p \end{cases}$;

(b) $EP[j] = g^{\leftarrow}(j)$ pour chaque $j \leqslant i$;

(c) pour tout $j < p$, $G_{\text{inverse}}[j] = \begin{cases} \max\left(\{l \leqslant i | g(l) = j\}\right) & \text{si un tel } l \text{ existe} \\ p & \text{sinon} \end{cases}$.

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 une 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 $j \leqslant i$. G_{inverse} contient toujours une inverse partielle de g sur $[\![0, j]\!]$ et est utilisé pour calculer EP.

12. Donc on a clairement besoin d'une fonction $i \mapsto n^K$. Sauf qu'il faut en limiter l'accès pour éviter de s'en servir de manière à obtenir des nombres trop gros ? Ou ce n'est pas notre problème parce que de toute façon, si le calcul dépasse les bornes imposées, on n'est plus dans la même classe de complexité ?
13. Est-ce toujours vrai pour une RAM polynomiale ?
14. Du coup on n'a plus besoin de la fonction $i \mapsto n^K$? Dans l'article, on a déjà accès à $cs.n$ pour simuler la machine. D'où vient cette connaissance ?

Voici l'algorithme pour Γ :

Input: s
// Initializations
$p := c(s.n)^K$;
$EP[0] := 0$;
for $j = 0$ *to* $s.n - 1$ **do**
  $F_{\text{in}}[j] := s.f(j)$;
end
for $j = s.n$ *to* $p - 1$ **do**
  $F_{\text{in}}[j] := 0$;
end
for $j = 0$ *to* $p - 1$ **do**
  $G[j] := j$;
  $G_{\text{inverse}}[j] := p$;
end
// Main loop
for $i = 0$ *to* $p - 1$ **do**
  $G[i] = \sigma[i]$;
  $EP[i] := \min(\{i, G_{\text{inverse}}[G[i]]\})$;
  $G_{\text{inverse}}[G[i]] := i$;
end
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-Predecesor* 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 $\mathcal{O}(p) = \mathcal{O}\left(n^K\right)$.

□

2.3.2 Calculable en temps polynomial ⇒ LSRS

CONJECTURE 4. *Si une fonction de RAM Γ est calculable en temps $\mathcal{O}\left(n^K\right)$ sur une $\{+,-\}$ [15]-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 [16] 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 jeu [17].

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 [2] 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.

$$f_0(x) = \begin{cases} 1 & \text{si } x = 0 \\ x + 1 & \text{si } 3x > 0 \\ f_0(x - 1) & \text{sinon} \end{cases} \tag{2.4}$$

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 :

$$A(x) = x \tag{2.1}$$
$$R'_A(x) = f(x) \tag{2.2}$$

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.

15. On aura probablement besoin de l'opération $\times$ si on a besoin d'avoir la fonction $i \mapsto n^K$.
16. Je ne comprends pas si ce $c(\Gamma)$ est le même que celui qu'on vient d'instancier.
17. C'est l'opération $g, g' \mapsto g' \left[g^{\leftarrow}(x)\right]_x$

Sur le domaine $[\![cn^K, 2cn^K - 1]\!]$:

$$I(x) = \begin{cases} 1 & \text{si } x = cn^K \\ I'(x-1) & \text{sinon} \end{cases} \tag{2.1}$$

$$I'(x) = \begin{cases} i & \text{si } I(x) \text{ est de la forme IF}(A = B)\{I(i)\} \text{ ELSE } \{I(j)\} \text{ et } A(x) = B(x) \\ j & \text{si } I(x) \text{ est de la forme IF}(A = B)\{I(i)\} \text{ ELSE } \{I(j)\} \text{ et } A(x) \neq B(x) \\ I(x) & \text{si } I(x) \text{ est de la forme HALT} \\ I(x)+1 & \text{sinon} \end{cases} \tag{2.2}$$

$$A(x) = \begin{cases} 0 & \text{si } x = cn^K \\ A'(x-1) & \text{sinon} \end{cases} \tag{2.3}$$

$$A'(x) = \begin{cases} c & \text{si } I(x) \text{ est de la forme } A := c \\ A(x) * B(x) & \text{si } I(x) \text{ est de la forme } A := A + B \\ R_A(x) & \text{si } I(x) \text{ est de la forme } A := R_A \\ N(x) & \text{si } I(x) \text{ est de la forme } A := N \\ A(x) & \text{sinon} \end{cases} \tag{2.4}$$

$$B(x) = \begin{cases} 0 & \text{si } x = cn^K \\ B'(x-1) & \text{sinon} \end{cases} \tag{2.5}$$

$$B'(x) = \begin{cases} A(x) & \text{si } I(x) \text{ est de la forme } B := A \\ B(x) & \text{sinon} \end{cases} \tag{2.6}$$

$$N(x) = \begin{cases} n & \text{si } x = cn^K \\ N'(x-1) & \text{sinon} \end{cases} \tag{2.7}$$

$$N'(x) = \begin{cases} A(x) & \text{si } I(x) \text{ est de la forme } N := A \\ N(x) & \text{sinon} \end{cases} \tag{2.8}$$

$$R_A(x) = R'_A\left[A^{\leftarrow}(x)\right]_x \tag{2.9}$$

$$R'_A(x) = \begin{cases} B(x) & \text{si } I(x) \text{ est de la forme } R_A := B \\ R_A(x) & \text{sinon} \end{cases} \tag{2.10}$$

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 $i_1, \ldots, i_m$ 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 $I\left[1^{\leftarrow}(x)\right]_x$ 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 $R(0), \ldots, R(n'-1)$, à la fin du calcul. Pour $2cn^K \leqslant x < 3cn^K$, le LSRS consiste en les équations suivantes :

$$A(x) = x - 2cn^K \tag{2.1}$$

$$R_A(x) = R'_A\left[A^{\leftarrow}(x)\right]_x \tag{2.2}$$

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, $1(-), n(-), id(-)$ [18]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 :

$$A(a) = a \tag{2.1}$$

$$R'_A(a) = s.f(a) \tag{2.2}$$

$$I(a) = B(a) = N(a) = R_A(a) = I'(a) = A'(a) = B'(a) = N'(a) = 0 \tag{2.3}$$

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 faire que M contient $s.f(i)$ dans le registre R_i.

18. On obtient la fonction $i \mapsto cn^K$ avec une astuce, cf cette note.

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 :

- $c(s.n)^K \leqslant A^{\leftarrow}(a) < a$. 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 $R'_A\left(A^{\leftarrow}(a)\right)$ 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 $f(A^{\leftarrow}(a))$ [19]. 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, $R_A\left(2c(s.n)^K + a\right)$ 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.

□

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

Chapitre 3

LSRS à arité multiple

3.1 Introduction

Avant toute chose, il peut être bon de rappeler à quoi ressemblent les LSRS. [5] [2]. Cependant, pour les besoins du présent chapitre, nous allons ajouter une troisième forme d'opération.

Définition 27 (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 $i \leqslant k$, 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* $* \in \{+, -, \times\}$ [1]
- *(récursion)* $f_i(x) = g'\left[g^{\leftarrow}(x)\right]_x$ *où* $g' \in F_k$ *et* $g \in F_{i-1}$
- *(composition)* $f_i(x) = g'\left[g(x)\right]_x$ *où* $g' \in F_k$ *et* $g \in F_{i-1}$

Remarque 2. L'ajout de cette opération facilitera les calculs qui suivront. Elle ne rend pas le LSRS plus puissant [2].

Le but de ce chapitre est d'étendre les LSRS définis dans [2] 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 l'ordre lexicographique permet de facilement définir un LSRS équivalent d'arité 1, ce qui n'apporte pas grand-chose par rapport à ce dont nous avons déjà parlé.

Ici, on ne va pas considérer de LSRS dont toutes les fonctions sont de même arité. Soit $a > 1$, on suppose que le LSRS est composé de fonctions f_i d'arité $r_i \leqslant a$, 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.

Dans la suite, on notera $(\leqslant a)$-uplet pour parler d'un n-uplet où $n \leqslant a$.

3.2 Les ennuis commencent

3.2.1 Bon ordre sur les $(\leqslant a)$-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 $(\leqslant a)$-uplets tels que $\bar{x} <_{\text{naïf}} \bar{y} \Leftrightarrow |\bar{x}| < |\bar{y}|$ 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 ressemble à une contrainte inutile [2].

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...

1. Les opérations peuvent varier, il est dit dans [2] que l'on peut choisir n'importe quelle opération binaire calculable en temps linéaire sur machine de Turing, voire la multiplication. Nous verrons d'ailleurs par la suite que, pour des raisons de facilité, nous aurons besoin de la multiplication.

2. Il y avait de plus une autre raison, mais je ne m'en souviens plus au moment où j'écris ces lignes. C'est apparemment un bon ordre (*tout ensemble non vide a un plus petit élément*), mais il y avait un contre-argument qui nous avait fait chercher un autre ordre. Parce qu'on atteint très vite les bornes du domaine ?

Définition 28. *On définit l'ordre $<$ sur les $(\leqslant a)$-uplets par :*

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

Si toutes les arités $\leqslant a$ 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 29. *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 3. Quelques remarques informelles :

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

3.2.2 Propriétés combinatoires

Proposition 30. *Soit $\bar{x}$ un $(\leqslant a)$-uplet. On suppose que toutes les arités sont possibles. Notons $m = \max(\bar{x})$ et $r = |\bar{x}|$.*
Le bon ordre $<$ sur les $(\leqslant a)$-uplets vérifie les égalités suivantes :

1. *$card(\{\bar{y} < \bar{x} | \max(\bar{y}) < \max(\bar{x})\}) = \sum_{i=1}^{a} m^i$* [3] *;*
2. *$card(\{\bar{y} < \bar{x} | \max(\bar{y}) = \max(\bar{x}) \wedge |\bar{y}| < |\bar{x}|\}) = \sum_{i=1}^{r} \left((m+1)^i - 1\right)$;*
3. *$card(\{\bar{y} < \bar{x} | \max(\bar{y}) = \max(\bar{x}) \wedge |\bar{y}| = |\bar{x}| \wedge \bar{y} <_{lex} \bar{x}\}) = \sum_{i=1}^{r} c_i$ où $c_i = x_i \times (m+1)^{r-i}$ si $x_{i-1} = m$ ou $x_{i-2} = m$ ... $x_1 = m$, 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 $(\leqslant a)$-uplet en choisissant une arité $i \leqslant a$ 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 $< m$, d'où le résultat.

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 défini [4] 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 $y_1 < x_1 \leqslant m$, 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 : $c_1 = x_1 \times \left((m+1)^{r-1} - m^{r-1}\right)$.

Une fois que $\sum_{j=1}^{i-1} c_j$ a été calculé pour $i \leqslant r$, 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 : $x_i \times \left((m+1)^{r-i} - m^{r-i}\right)$.

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

Corollaire 31. *Le rang de $\bar{x}$ est $\left(\left(\sum_{i=1}^{a} m^i\right) + \left(\sum_{i=1}^{r} \left((m+1)^i - 1\right)\right) + \left(\sum_{i=1}^{r} c_i\right)\right)$ où $c_i = x_i \times (m+1)^{r-i}$ si $x_{i-1} = m$ ou $x_{i-2} = m$ ou ... ou $x_1 = m$, et $c_i = (m+1)^{r-i} - m^{r-i}$ sinon.*

3. J'ai mis -1 ici au début, et j'avais l'air convaincu, mais je ne sais plus pourquoi...
4. Attention : ε_i ne prend en compte que les coordonnées x_1 à x_{i-1}.

3.3 Déroulement d'un calcul de a-LSRS

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 déroule de la façon suivante :

- $\bar{x} = (0)$: 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.
- $\bar{x} = (0,0) = s((0))$: 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 $j < i$;
- $s(\bar{x})$:
  - 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 Ce qui suit est la piste principale que nous suivons en ce moment. Le LSRS pourrait être élargi de bien des manières, mais c'est celle-ci qui nous semble la plus intéressante et la plus naturelle, une fois que l'on a choisi comment ordonner les ($\leqslant a$)-uplets.

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 : $\bar{x'} < \bar{x}$, $\left|\bar{x'}\right| < |\bar{x}|$ et $\forall j\ \exists j'\ x'_j = x_{j'}$ [5].

Définition 32 (a-LSRS)**.** *Soit F un ensemble de symboles de fonctions de base. Soient $f_1, \ldots, f_k$ de nouveaux symboles de fonctions n'apparaissant pas dans F, d'arités respectives $1 \leqslant r_1 \leqslant r_2 \leqslant \cdots \leqslant r_k = a$ [6]. 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 $G_i = F \cup \{f_j | r_j < r_i\}$ [7].*

Un a-LSRS S sur F et $f_1, \ldots, f_k$ est une suite d'équations $E_1, \ldots, E_k$ 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) $f_i(\bar{x}) = g'\left[g^{\leftarrow}(\bar{x'})\right]_{\bar{x'}}$, où arité$(g)$ = 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 4. L'opération $g'\left[g^{\leftarrow}(\bar{x'})\right]_{\bar{x'}}$ nécessite une brève redéfinition, puisqu'on ne peut plus renvoyer l'unique variable contrairement à précédemment.

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

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 5. 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 $\pi^i_j(-)$, 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.

5. 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 $\left|\bar{x'}\right| < |\bar{x}|$. 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).

6. 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.

7. 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.

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 33 (RAM-structure)**.** *Soit t un a-type, c'est-à-dire une signature fonctionnelle contenant des fonctions d'arité $\leqslant a$ 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 $C \in t$;*
- *$f : n \times \cdots \times n \to \mathbb{N}$ 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 $\bar{x} \in n \times \cdots \times n$.

Définition 34 (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 [8]*.*
On dit que Γ est polynomiale lorsque $\Gamma(s).n = \mathcal{O}\left((s.n)^K\right)$, et linéaire lorsque $\Gamma(s).n = \mathcal{O}\,(s.n)$.

Définition 35 (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 $f_1, \ldots, f_k$ 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 $f_1, \ldots, f_k : cs.n \times \cdots \times cs.n \to (cs.n)^a$ 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.

3.4 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.

Conjecture 5. *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 36. *Il existe un 1-LSRS S qui, à une variable x, associe le $(\leqslant a)$-uplet $\bar{x}$ de rang x.*

Démonstration. Pour simplifier, on va de nouveau se placer dans le cas où toutes les arités $\leqslant a$ sont représentées [9]. On va se servir du codage du rang donné en proposition 30, montrer qu'on peut le décoder par LSRS.

Calcul du max m. Considérons le LSRS suivant :

$$f_1(x) = \begin{cases} 0 & \text{si } x < a \\ 1 & \text{si } x = a \\ f_1(x-1) & \text{si } x < f_{4a-1}(x-1) \\ f_1(x-1)+1 & \text{si } x = f_{4a-1}(x-1) \end{cases} \tag{3.1}$$

$$f_{i_1+1}(x) = f_{i_1}(x) \times f_1(x) \tag{3.2}$$

$$f_{a+1}(x) = f_1(x) + f_2(x) \tag{3.3}$$

$$f_{a+i_2+1}(x) = f_{a+i_2}(x) + f_{i_2+2}(x) \tag{3.4}$$

$$f_{2a}(x) = f_1(x) + 1 \tag{3.5}$$

$$f_{2a+i_3+1}(x) = f_{2a+i_3}(x) \times f_{2a}(x) \tag{3.6}$$

$$f_{3a}(x) = f_{2a}(x) + f_{2a+1}(x) \tag{3.7}$$

$$f_{3a+i_4+1}(x) = f_{3a+i_4}(x) + f_{2a+i_4+2}(x) \tag{3.8}$$

8. 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

9. 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.

On rappelle que $f_1(x-1) = f_1\left[1^{\leftarrow}(x)\right]_x$ et que la distinction de cas ne rend pas le LSRS plus puissant [2].
Expliquons les équations.
(3.1) est censée calculer m, donc le max courant. Sachant cela, les équations suivantes coulent de source :

- (3.2) calcule m^{i_1} pour $i_1 \in [\![1, a]\!]$. On a aussi $f_a(x) = m^a$.
- (3.3) calcule $\sum_{i=1}^{2} m^i$ et (3.4) calcule $\sum_{i=1}^{i_2+2} m^i$ pour $i_2 + 2 \leqslant a$. 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 $(\leqslant a)$-uplets $(0, \ldots, 0)$ jusqu'à l'arité a, donc on énumère les a premiers $(\leqslant a)$-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 $(\leqslant a)$-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.

$$f_{4a}(x) = \begin{cases} 1 & \text{si } x = 0 \\ f_{4a}(x-1) & \text{si } x < f_{8a+2}(x-1) \\ 1 & \text{si } x = f_{4a-1}(x-1) \\ f_{4a}(x-1)+1 & \text{si } x = f_{8a+2}(x-1) \end{cases} \tag{3.9}$$

$$f_{i_1+4a}(x) = \begin{cases} (m+1)^{i_1} - 1 & \text{si } 1 \leqslant i_1 \leqslant f_{4a}(x) \leqslant a \\ 0 & \text{sinon} \end{cases} \tag{3.10}$$

$$f_{i_2+5a}(x) = \sum_{i=1}^{i_2} f_{i+4a}(x) \tag{3.11}$$

$$= \sum_{i=1}^{\min(f_{4a}(x), i_2)} \left((m+1)^i - 1\right) \tag{3.12}$$

$$f_{6a+1}(x) = \begin{cases} f_{4a}(x)+1 & \text{si } f_{4a}(x) < a \\ 1 & \text{sinon} \end{cases} \tag{3.13}$$

$$f_{i_3+6a+1}(x) = \begin{cases} (m+1)^{i_3} - 1 & \text{si } 1 \leqslant i_3 \leqslant f_{6a+1}(x) \leqslant a \\ 0 & \text{si } f_{6a+1}(x) = 1 = i_3 \\ 0 & \text{sinon} \end{cases} \tag{3.14}$$

$$f_{i_4+7a+1}(x) = \sum_{i=1}^{i_4} f_{i+6a+1}(x) \tag{3.15}$$

$$= \begin{cases} \sum_{i=1}^{\min(f_{6a+1}(x), i_4)} \left((m+1)^i - 1\right) & \text{si } 1 < f_{6a+1}(x) \\ 0 & \text{sinon} \end{cases}$$

$$f_{8a+2}(x) = f_{8a+1}(x) + f_{4a}(x) \tag{3.16}$$

On va un peu plus vite : les équations (3.10), (3.11), (3.14) et (3.15) 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 $f_{6a+1}(x)$-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.

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.

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

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 $(x_1, \ldots, x_{r-1}, m)$ 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. [10]

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 $x_1, \ldots, x_r$ est représenté par $f_1(x), \ldots, f_r(x)$. La simulation se fait dans les fonctions $f'_1(x), \ldots, f'_r(x)$.

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 :

$$f'_r(x) = \begin{cases} f_r(x-1)+1 & \text{si } f_r(x-1) < m \\ 0 & \text{si } f_r(x-1) = m \end{cases} \tag{3.17}$$

$$f'_{i-1}(x) = \begin{cases} f'_{i-1}(x-1)+1 & \text{si } f'_i(x) = 0 \text{ et } f'_i(x-1) = m \text{ et } f'_{i-1}(x-1) < m \\ 0 & \text{si } f'_i(x) = 0 \text{ et } f'_i(x-1) = m \text{ et } f'_{i-1}(x-1) = m \\ f'_{i-1}(x-1) & \text{sinon} \end{cases} \tag{3.18}$$

$$\text{future\_max}(x) = \max\left(f'_r(x), \ldots, f'_1(x)\right) \tag{3.19}$$

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 :

$$f_r(x) = \begin{cases} f_r(x-1)+1 & \text{si } f_r(x-1) < m \\ f_r(x-1) & \text{si } f_r(x-1) = m \text{ et “\textit{le max n'apparaît pas au tour suivant}”} \\ 0 & \text{si } f_r(x-1) = m \text{ et “\textit{le max apparaît au tour suivant}”} \end{cases} \tag{3.20}$$

$$f_{i-1}(x) = f'_{i-1}(x) \tag{3.21}$$

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 : $\text{future\_max}(x) = m$. 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 $i \leqslant r$, 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.

10. Je sais que pour l'instant ce n'est que du blabla, mais je ne me suis pas encore demandé comment le prouver. De plus, je pense qu'il s'agit plus d'une compréhension du fonctionnement de l'ordre. Mais il faudrait quand même le prouver.

- 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.
- Le premier r-uplet d'une arité r donnée est toujours de la forme $(0, \ldots, 0, m)$.
- Le premier $(\leqslant a)$-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 $\left|\bar{x'}\right| < |\bar{x}|$. Une fois qu'on a le $(\leqslant a)$-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 37. *Si la fonction $\bar{x} \mapsto \bar{x'}$ où $\bar{x'} \ll \bar{x}$ est simulable par LSRS, alors la fonction $\bar{x} \mapsto x'$, où x' est le rang de $\bar{x'}$, 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 :

$$\text{rang}\left(\bar{x'}\right) = \left(\sum_{i=1}^{a} m'^{i}\right) + \left(\sum_{i=1}^{r'} \left(\left(m'+1\right)^{i} - 1\right)\right) + \left(\sum_{i=1}^{r} c_i\right) \tag{3.22}$$

où $c_i = x'_i \times \left(m'+1\right)^{r'-i}$ si $x'_{i-1} = m'$ ou $x'_{i-2} = m'$ … $x'_1 = m'$, et $c_i = \left(m'+1\right)^{r'-i} - m'^{r'-i}$ 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 36. Le calcul des c_i n'est pas plus compliqué mais j'ai la flemme pour l'instant. □

LEMME 38. *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 $f\left(\bar{x}\right) = g_1\left(\bar{x_1}\right) * g_2\left(\bar{x_2}\right)$, pour $* \in \{+, -, \times\}$ et d'un a-LSRS 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 $(\leqslant a)$-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 [11]simule $f\left(\bar{x}\right) = g_1\left(\bar{x_1}\right) * g_2\left(\bar{x_2}\right)$:

$$f_1(x) = \hat{g_1}\left[\text{sub}_1(x)\right]_x \tag{3.23}$$
$$f_2(x) = \hat{g_2}\left[\text{sub}_2(x)\right]_x \tag{3.24}$$
$$f^1(x) = f_1(x) * f_2(x) \tag{3.25}$$

□

Remarque 6. Avant de traiter l'opération suivante, il faut apporter une précision. Soit le 2-LSRS suivant :

$$f_1(x) = n(x) + 1(x) \tag{3.26}$$
$$f_2(x,y) = f_1(x) + 1(y) \tag{3.27}$$
$$f_3(x,y) = f_1(x) + id(y) \tag{3.28}$$

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 : $(0) < (0,0) < (1) < (0,1) < (1,0) < (1,1) < (2) \ldots$. Soit x un $(\leqslant 2)$-uplet, notons $r(x)$ 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,

11. La numérotation est disjointe de celles précédemment données.

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 :

$$\begin{aligned}
\text{arité}_1(x) &= 1(x) && (3.29)\\
&\dots\\
\text{arité}_a(x) &= \text{arité}_{a-1}(x) + 1(x) && (3.30)\\
&\dots\\
\text{a_c}(x) &= \text{arité courante} && (3.31)\\
&\dots\\
\hat{f}(x) &= \begin{cases} \hat{\sigma}(\bar{x}) & \text{si a_c}(x) = \text{arité}_r(x) \\ \hat{f}(x-1) & \text{sinon} \end{cases} && (3.32)
\end{aligned}$$

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 39. *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 $f(\bar{x}) = g_1\left[g_2^{\leftarrow}(\bar{x'})\right]_{\bar{x'}}$ 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 $g_1\left[g_2^{\leftarrow}(\bar{x})\right]_{\bar{x}}$. Rappelons la définition donnée plus haut :

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

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	...
$f_1\left[f_2^{\leftarrow}(x)\right]_x$	0	ND	1	ND	ND	ND	2	...
$f_1\left[f_3^{\leftarrow}(x)\right]_x$	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 première simulation 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	...

Premièrement, la remarque précédente impose que les fonctions renvoient leur dernière valeur quand leur équivalente n'est pas définie. Je ne sais pas comment le dire, mais cette condition permet d'assurer que l'equal-predecesor fonctionne correctement. Ici, l'equal-predecesor revient à revenir en arrière sur une même 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-predecesor et on regarde quelle valeur on trouve dans cette case ; c'est ce que renvoie l'equal-predecesor. 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.

Deuxièmement, $f_1\left[f_2^{\leftarrow}(x)\right]_x$ 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 :

$$g_1(x) = \mathrm{id}\left[\hat{f}_2{}^{\leftarrow}(x)\right]_x = \begin{cases} \max\left(\left\{y < x | \hat{f}_2(x) = \hat{f}_2(y)\right\}\right) & \text{si de tels } y \text{ existent} \\ x & \text{sinon} \end{cases} \tag{3.33}$$

$$g(x) = \begin{cases} \hat{f}_1\left[\hat{f}_2{}^{\leftarrow}(x)\right]_x & \text{si } g_1(x) < x \\ x_i & \text{sinon} \end{cases} \tag{3.34}$$

Rappelons que l'on a par défaut un LSRS qui permet de récupérer les coordonnées du $(\leqslant a)$-uplet de rang x.
Et il manque une preuve, mais il faut faire ça pour la simulation.
On se ramène au cas précédent en utilisant $f'(\bar{y}) = g_1\left[g_2^{\leftarrow}(\bar{y})\right]_{\bar{y}}$ et $f(\bar{x}) = f'\left(\bar{x'}\right)$, où $|\bar{y}| = \left|\bar{x'}\right|$.

□

Appendices

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

.1.1 Conventions d'écriture

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

$$\text{(fn) } s_a\text{NOM\_FONCTION(arguments)}s_b$$

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

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

Par exemple :

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

doit se comprendre :

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

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

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

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

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

doivent se comprendre :

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

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

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

.1.2 Ecriture des programmes

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

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

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

Input: μ, π_1
// Si μ est un terme de mémoire non vide, on extrait la première valeur de cette mémoire en laissant le reste inchangé.
if *le terme contenu dans* μ *est de la forme* $MEM(i, A, m)$ **then**
  Stocker A dans π_1 ;
  Remplacer le contenu de μ par m ;
end

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

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

Input: μ, π_1
// Si μ est un terme de mémoire non vide, on extrait la première valeur de cette mémoire en laissant le reste inchangé.
(switch) $s_a\mu s_f s_{\mathrm{MEM}}$;
(2-dest) $s_{\mathrm{MEM}}\mu\pi_1 s_b$;
(3-dest) $s_b\mu\mu s_f$;

Algorithm 10: Fonction $s_a\mathrm{EXTRACT\_FIRST}(\mu, \pi_1)s_f$. Algorithme plus haut.

Cette fonction va de pair avec la fonction DECOMPOSE.

Input: $\alpha, \pi, \pi', \bar{\pi}$
// 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 ;
foreach *sous-terme de* α **do**
  Stocker le sous-terme dans π';
  Construire dans π le terme $\mathrm{MEM}(\pi', \pi', \pi)$;
end
Copier α dans π' ;

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

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

Input: $\alpha, \pi, \pi', \bar{\pi}$
// Détruit le terme contenu dans 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) $s_a \alpha s_{a_1} \dots s_{a_k,j}$;
for i *from* 1 *to* k **do**
  for j *from* r_i *to* 1 **do**
    (j-dest) $s_{a_{i-1},j} \alpha \pi' s_{b_{i-1},j}$;
    (const) $s_{b_{i-1},j} \pi' \pi' \pi \mathrm{MEM} \pi s_{a_{i-1},j+1}$;
    (correction) $s_{a_{i-1},r_i+1} = s_{a_i,1}$;
  end
  (correction) $s_{a_k,1} = s_b$;
end
(fn) $s_b \mathrm{COPY}(\alpha, \pi', \bar{\pi}) s_c$;

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

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

Input: $\alpha, \beta, s_1, s_0, \pi_1, \pi_2, \pi'_1, \pi'_2$
/* 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. */
/* */
/* Vérifie si $\alpha = \beta$ en faisant une analyse inductive sur la construction des termes qu'ils contiennent. */

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

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

(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 α et β, on les stocke dans π_1 et π_2, et on récupère le premier sous-terme dans π'_1 et π'_1.

(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 π_1, π_2 pour le mettre dans π'_1, π'_2.

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

(fn) s_gDECOMPOSE$(\pi'_1, \pi'_1, \pi_1) s_{g_1}$;
(fn) s_{g_1}DECOMPOSE$(\pi'_2, \pi'_2, \pi_2) s_{g_2}$;
// On récupère les sous-termes de π'_1 et π'_2, on les stocke dans π_1 et π_2, et on remplace π'_1 et π'_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 ($s_{g_2} \to s_{g_3} \to s_1$) 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 ($s_{g_2} \to s_{g_3} \to s_0$ ou $s_{g_2} \to s_{g_4} \to s_0$) alors les deux termes n'étaient pas égaux. Enfin, si les deux mémoires ont encore des termes à vérifier, alors on revient à l'état s_{d_1}.

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

Input: $\mu, \mu_1, \mu_2, \mu_3, \mu_4, \alpha, \beta, \pi_1, \pi_2, \pi'_1, \pi'_2$

/* $\mu, (\mu_i)_{i\in 4}$ sont tels que décrits plus haut. */
/* α est l'indice auquel on veut insérer. */
/* β est la valeur qu'on veut insérer à l'indice α. */
/* $\pi_1, \pi_2, \pi'_1, \pi'_2$ 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)$s_{a_4} \mu_1 \mu_2 \mu_3 \mathrm{MEM} \mu_3 s_b$;

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

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

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

(const)$s_{\mathrm{false}} \mu_1 \mu_2 \mu_4 \mathrm{MEM} \mu_4 s_{f_1}$;
(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)$s_{c,\varepsilon} \mu_1 \mu_2 \mu_3 \mathrm{MEM} \mu_3 s_{\mathrm{copie}}$;
(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)$s_{m_3} \mu_1 \mu_2 \mu_3 \mathrm{MEM} \mu_3 s_{\mathrm{copie}}$;
(fn)$s_{\mathrm{copie}} \mathrm{COPY}\,(\mu_3, \mu, \mu_1, \mu_2, \mu_4)\, s_f$;

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

Input: $\mu, \mu_1, \mu_2, \mu_3, \mu_4, \alpha, \beta, \pi_1, \pi_2, \pi'_1, \pi'_2$

/* $\mu, (\mu_i)_{i\in 4}$ sont tels que décrits plus haut. */
/* α est l'indice auquel on veut accéder. */
/* β est le registre dans lequel on veut stocker la valeur d'indice α. */
/* $\pi_1, \pi_2, \pi'_1, \pi'_2$ 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)$s_{a_4} \mu_1 \mu_2 \mu_3 \mathrm{MEM} \mu_3 s_b$;

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

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

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

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

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

Bibliographie

[1] Stephen Bellantoni and Stephen Cook. A new recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2(2) :97–110, 1992.

[2] E. Grandjean and T. Schwentick. Machine-independent characterizations and complete problems for deterministic linear time. *SIAM Journal on Computing*, 32(1) :196–230, 2003. cited By 8.

[3] Etienne Grandjean. Invariance properties of rams and linear time. *Computational Complexity*, 4(1) :62–106, 1994.

[4] 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.

[5] 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.