# Fondements de la Programmation

CM: Paulin de Naurois - TD: Paulin de Naurois, Lê Thành Dũng Nguyễn

Notes de Cours (P. Boudes, V. Mogbil, P. de Naurois.)

### Bilbiographie

Ces notes de cours utilisent le livre de de Neil Jones "Computability and Complexity: from a Programming Perspective" et celui de P. Clote / E. Kranakis "Boolean Functions and Computation Models" pour les modèles de calculs, le livre de Barendregt "The Lambda Calculus, Its Syntax and Semantics" et les notes de cours de J. Goubault-Larrecq, de T. Hardin et de Y. Bertot pour le $\lambda$-calcul.

## 1 Introduction

### Thèse de Church-Turing

Toutes les formalisations raisonnables de la notion de calcul et d'algorithme sont équivalentes.

*Remarque essentielle:* On distingue deux notions:

- Une *fonction*, qui, à chaque entrée (par exemple un entier naturel), associe une valeur (par exemple un entier naturel).
- Un algorithme, qui est une description précise de la façon de calculer un résultat sur une entrée donnée. Un programme est la représentation d'un algorithme dans un langage de programmation fixé.

Un algorithme réalise une fonction partielle: la valeur associée à chaque entrée est le résultat du calcul de l'algorithme sur cette entrée, si l'algorithme termine. Une fonction peut être réalisée par plusieurs algorithmes différents, ou par aucun. L'ensemble des algorithmes de $\mathbb{N} \to \mathbb{N}$ est dénombrable : il suffit pour s'en persuader de considérer leur écriture sur un alphabet fini. L'ensemble des fonctions de $\mathbb{N} \to \mathbb{N}$ est quant à lui indénombrable - plus précisément, en bijection avec $\mathbb{R}$. Il existe donc infiniment plus de fonctions que d'algorithmes, et seule une infime proportion de ces fonctions est réalisable par des algorithmes.

La Thèse de Church-Turing peut être reformulée comme suit: Quel que soit le formalisme raisonnable choisi pour décrire les algorithmes, les fonctions réalisées sont les mêmes. Pour prouver cette thèse de Church-Turing pour deux formalismes de description des algorithmes, il faut donner une méthode systématique de traduction des instructions de programmation d'un formalisme dans le second: cette méthode systématique est elle-même un algorithme, qu'on appelle *compilateur*.

### Modèles de calcul

Voici quelques modèles abstraits de calcul que l'on va étudier :

- Machine abstraite de Turing (MT) ,
- Machine abstraite à compteurs (MC), à adressage indirect (SRAM),
- Machine abstraite parallèles (PRAM),

- Programmation par fonctions récursives (Kleene)
- Programmation fonctionnelle: $\lambda$-calcul (Church)

Pour tout ces formalismes, la thèse de Church-Turing est vérifiée. Ces modèles sont très différents mais certains ont des aspects communs : un algorithme est un ensemble fini d'instructions, l'exécution est faite pas-à-pas de façon déterministe, en utilisant une quantité finie de mémoire ou de temps, pour lesquels il n'y a pas de borne a priori sur leur quantité.

## Notations Communes

On adopte les notations suivantes pour les modèles impératifs (MT - MC - SRAM - PRAM), issues de livre de N. Jones.

- $M - data$ décrit la structure de donnée des entrées et des sorties - i.e le domaine et le co-domaine des fonctions calculées. Dans tous les modèles de calcul considérés dans le cadre de ce cours, ces domaines et co-domaines sont en bijection avec $\mathbb{N}$.
- $M - store$ décrit la structure de données *interne* à la machine: c'est une description de sa mémoire. Cette structure dépend du modèle choisi.
- $M - prog$ décrit le langage d'instructions du modèle.

Pour chacun de ces modèles un programme $P$ est défini comme une suite finie d'instructions, numérotées. La dernière instruction est vide, et le programme termine lorsqu'il effectue cette instruction vide. Afin de faciliter la rédaction des programmes, on autorise également l'utilisation d'étiquettes explicites, en plus des numéros d'instruction. Dans l'exemple ci-dessous, la dernière instruction est étiquetée par le mot "FIN".

$$\begin{array}{rl} P = & 1 : I_1, \\ & 2 : I_2, \\ & \vdots \\ & m : I_m, \\ \text{FIN} & m+1 : \end{array}$$

**Exécutions** Sur une donnée d'entrée fixée, la mémoire de la machine est initialisée avec l'entrée du programme. Cette initialisation est donnée par la fonction $Readin : M - data \to M - store$, qui décrit l'état initial de la machine sur l'entrée spécifiée. A la fin de l'exécution, le résultat du calcul est lu dans la mémoire de la machine. Cette lecture du résultat est donnée par la fonction $Readout : M - store \to M - data$. Pendant l'exécution du programme, la machine passe d'un état au suivant, par le biais de *transitions*. Chaque état de la machine est donné par un couple $(l, \sigma)$. Dans cet état $(l, \sigma)$, $l$ désigne l'instruction courante du programme ($l \in \{1, \cdots, m+1\}$ dans ce cas) pour les MT, MC et SRAM. Pour les PRAM, $l$ désigne l'instruction courante du programme de chaque processeur ($l$ est alors une fonction de $\mathbb{N}^+$ vers $\{1, \cdots, m+1\}$). De plus, dans un état $(l, \sigma)$, $\sigma \in M - store$ est la mémoire. On note une transition d'un état au suivant comme suit: $p \vdash (i, \sigma_i) \to (j, \sigma_j)$.

Sauf pour certains cas spécifiques d'instructions de saut (conditionnel ou non), une transition depuis un état $(i, \sigma)$ passe à un état $(i+1, \sigma')$, c'est-à-dire que la prochaine instruction à être exécutée est celle qui suit dans le programme.

On note $\to^*$ la clôture réflexive transitive de la relation de transition, c'est-à-dire $(i_1, \sigma_1) \to^* (i_n, \sigma_n)$ si et seulement si il existe une suite de longueur finie, possiblement nulle, de transitions $(i_1, \sigma_1) \to (i_2, \sigma_2 \to \ldots \to (i_n, \sigma_n)$.

L'effet d'un programme $p$ sur l'entrée x est alors $[p](x) = y$ si et seulement si

1. $\sigma_0 = Readin(x)$
2. $p \vdash (1, \sigma_0) \to^* (m+1, \sigma)$
3. $y = Readout(\sigma)$

# 2 Machines de Turing avec langage d'instructions (MT)

On travaille sur un alphabet fini, en général binaire $\{0,1\}$. La machine dispose d'un ruban bi-infini pour mémoriser l'entrée et faire les calculs. Il s'agit d'une suite infinie de cellules contenant chacune un symbole de l'alphabet, ou le symbole Blanc ($B$); seul un nombre fini de cellules contiennent un symbole non-blanc. On dispose d'une tête de lecture/écriture pour lire/modifier le symbole courant, et d'instructions de contrôle. Le langage d'instruction permet de déplacer la tête de lecture/écriture sur le ruban d'une cellule, à gauche ou à droite. On peut écrire un symbole sur le ruban, et tester quel est le symbole courant. La convention d'initialisation est que le ruban ne contienne que des symboles B (blanc) sauf à droite de la tête de lecture/écriture où l'on a l'entrée. Il en est de même pour le résultat lorsque l'exécution est terminée.

## 2.1 Synthèse

- $MT - data = \{0,1\}^*$ (les mots finis sur l'alphabet $\{0,1\}$).
- $MT - store = \{B^l.L.\underline{S}.R.B^r | L, R : String\ S : Symbol\}$, où la position de la tête de lecture/écriture est la cellule soulignée, avec:

$$
\begin{array}{rcll}
B^l & = & B^l.B & \text{(Autrement dit, } B^l \text{ est une suite infinie vers la gauche de symboles } B) \\
B^r & = & B.B^r & \text{(Autrement dit, } B^r \text{ est une suite infinie vers la droite de symboles } B) \\
L, R : String & ::= & S\ String \mid \epsilon \text{ (mot vide)} & \\
 & & \multicolumn{2}{l}{\text{(Autrement dit, } String \text{ désigne les suites finies de symboles)} } \\
S, S' : Symbol & ::= & 0 \mid 1 \mid B & \\
 & & \multicolumn{2}{l}{\text{(Autrement dit, les symboles sont 0, 1 et } B)}
\end{array}
$$

- $Readin(x) = B^l\underline{B}xB^r$, avec $B^l = B^l.B$ et $B^r = B.B^r$
- $Readout(B^l\underline{B}yB^r) = y$: le résultat du calcul est le mot lu à partir de la première case à droite de la tête de lecture, jusqu'au premier symbole $B$. Toutes les autres cases contiennent des symboles $B$.
- $MT - prog ::= right \mid left \mid write\ S \mid if\ S\ goto\ l'\ else\ l''$
  L'instruction $right$ déplace la tête de lecture d'une case à droite. L'instruction $left$ déplace la tête de lecture d'une case à gauche. L'instruction $if\ S\ goto\ l'\ else\ l''$ change l'instruction à exécuter à $l'$ si le symbole courant est $S$, et à $l''$ sinon.

**Remarque** lorsque l'on effectue une transition depuis l'instruction $if\ S\ goto\ l'\ else\ l'$, avec deux numéros identiques $l'$ d'instruction pour les deux cas du branchement, l'instruction suivante est dans tous les cas $l'$. On s'autorise donc à utiliser une instruction $goto\ l'$ à la place de $if\ S\ goto\ l'\ else\ l'$.

**Variantes** Dans le cas où on veut représenter une fonction qui prend deux entrées (par exemple, la fonction qui teste l'égalité de deux mots), la notation devient: $Readin(x_1, x_2) = B^l\underline{B}x_1Bx_2B^r$: les deux mots $x_1$ et $x_2$ sont donnés sur le ruban, séparés par un symbole $B$. De la même manière, pour représenter une fonction qui retourne deux valeurs, la notation devient: $Readout(B^l\underline{B}y_1By_2B^r) = (y_1, y_2)$. On peut généraliser de la sorte pour n'importe quel nombre de valeurs à donner en entrée ou à lire en sortie.

## 2.2 Transitions

| | | | |
|---|---|---|---|
| $p \vdash (l, B^l \underline{S} S' B^r)$ | $\rightarrow$ | $(l+1, B^l S \underline{S'} B^r)$ | si $I_l = right$ |
| $p \vdash (l, B^l \underline{S})$ | $\rightarrow$ | $(l+1, B^l S \underline{B})$ | si $I_l = right$ |
| $p \vdash (l, B^l S \underline{S'} B^r)$ | $\rightarrow$ | $(l+1, B^l \underline{S} S' B^r)$ | si $I_l = left$ |
| $p \vdash (l, \underline{S} B^r)$ | $\rightarrow$ | $(l+1, \underline{B} S B^r)$ | si $I_l = left$ |
| $p \vdash (l, B^l \underline{S} B^r)$ | $\rightarrow$ | $(l+1, B^l \underline{S'} B^r)$ | si $I_l = write\ S'$ |
| $p \vdash (l, B^l \underline{S} B^r)$ | $\rightarrow$ | $(l', B^l \underline{S} B^r)$ | si $I_l = if\ S\ goto\ l'\ else\ l"$ |
| $p \vdash (l, B^l \underline{S'} B^r)$ | $\rightarrow$ | $(l", B^l \underline{S} B^r)$ | si $I_l = if\ S\ goto\ l'\ else\ l"$ |

## 2.3 Exemple

La machine qui échange les symboles 0 et 1 sur le mot d'entrée peut s'écrire comme suit:

$$
\begin{array}{rll}
P = & & 1: \ right, \\
 & & 2: \ if\ B\ goto\ P2\ else\ 3, \\
 & & 3: \ if\ 1\ goto\ Cas1\ else\ Cas0, \\
 & Cas1 & 4: \ write\ 0, \\
 & & 5: \ goto\ 1, \\
 & Cas0 & 6: \ write\ 1, \\
 & & 7: \ goto\ 1, \\
 & P2 & 8: \ left, \\
 & & 9: \ if\ B\ goto\ FIN\ else\ 8, \\
 & FIN & 10:
\end{array}
$$

Exécutons le programme $P$ sur l'entrée $x = 1.0.1$. On a alors $Readin(x) = L\underline{B}101B^r$. La suite de transitions est

$$
\begin{array}{lllllllllll}
(1, B^l \underline{B}101B^r) & \rightarrow & (2, B^l B\underline{1}01B^r) & \rightarrow & (3, B^l B\underline{1}01B^r) & \rightarrow & (4, B^l B\underline{1}01B^r) & \rightarrow & (5, B^l B\underline{0}01B^r) & \rightarrow \\
(1, B^l B\underline{0}01B^r) & \rightarrow & (2, B^l B0\underline{0}1B^r) & \rightarrow & (3, B^l B0\underline{0}1B^r) & \rightarrow & (6, B^l B0\underline{0}1B^r) & \rightarrow & (7, B^l B0\underline{1}1B^r) & \rightarrow \\
(1, B^l B0\underline{1}1B^r) & \rightarrow & (2, B^l B01\underline{1}B^r) & \rightarrow & (3, B^l B01\underline{1}B^r) & \rightarrow & (4, B^l B01\underline{1}B^r) & \rightarrow & (5, B^l B01\underline{0}B^r) & \rightarrow \\
(1, B^l B01\underline{0}B^r) & \rightarrow & (2, B^l B010\underline{B}R) & \rightarrow & (8, B^l B010\underline{B}R) & \rightarrow & (9, B^l B01\underline{0}B^r) & \rightarrow & (8, B^l B01\underline{0}B^r) & \rightarrow \\
(9, B^l B0\underline{1}0B^r) & \rightarrow & (8, B^l B0\underline{1}0B^r) & \rightarrow & (9, B^l B\underline{0}10B^r) & \rightarrow & (8, B^l B\underline{0}10B^r) & \rightarrow & (9, B^l \underline{B}010B^r) & \rightarrow \\
(10, B^l \underline{B}010B^r). & & & & & & & & &
\end{array}
$$

L'instruction 10 étant vide, l'exécution s'arrête. Le résultat du calcul est alors $Readout(B^l \underline{B}010B^r) = 0.1.0$.

## 2.4 Machine de Turing avec langage d'instructions, à $k$ rubans

Le modèle de la machine de Turing se généralise à plusieurs rubans. Dans ce cas, seul le premier ruban est utilisé pour la lecture de l'entrée et du résultat du calcul. Les instructions sont les mêmes que précédemment, mais propres à *chaque* ruban.

- $MT^k - data = \{0,1\}^*$
- $MT^k - store = Tape^k$, où $Tape = \{B^l.L.\underline{S}.R.B^r | L, R : String\ S : Symbol\}$, avec les mêmes notations que pour les MT à un ruban.
- $Readin(x) = (B^l \underline{B}xB^r, B^l \underline{B}BB^r, \cdots, B^l \underline{B}BB^r)$,
- $Readout(B^l \underline{B}yB^r, Tape_2, \cdots, Tape_k) = y$, i.e. le mot binaire commençant à droite de la tête de lecture du premier ruban, jusqu'au premier symbole $B$. Le contenu des autres rubans est ignoré.
- $MT - prog ::= right_i \mid left_i \mid write_i\ S \mid if_i\ S\ goto\ l'\ else\ l"$

### 2.4.1 Un ou plusieurs rubans?

Toute fonction calculable par une Machine de Turing à un ruban est évidemment calculable par une machine de Turing à plusieurs rubans: il suffit d'utiliser le premier ruban seulement. La réciproque est vraie également, mais un peu plus compliquée à appréhender. Pour simuler une machine M2 à deux rubans avec une machine M1 à un seul ruban, il faut représenter les données de la mémoire de M2 avec un seul ruban. Cela peut se faire en *entrelaçant* les deux rubans de M2: on écrit sur un seul ruban, successivement, un symbole du ruban 1 de M2, puis un symbole du ruban 2 de M1. Ainsi, les données des deux rubans (bi-infinis) de M2 sont représentées en un seul ruban bi-infini, sur le même alphabet. Mais ce n'est pas suffisant: M1 ne disposant que d'une tête de lecture, ne peut pas simuler sans information supplémentaire les positions des deux têtes de lecture de M2. Il faut donc également représenter sur le ruban de M1 les positions des deux têtes de lecture de M2. Cela se fait en posant deux drapeaux sur les cases du ruban de M1 représentant les têtes de ruban de M2. Poser un drapeau revient en l'occurrence à entrelacer le ruban avec un autre ruban, bi-infini, contenant des blancs partout sauf en un seul endroit, correspondant à la position de la tête de lecture. Ainsi, les deux rubans $B^l\underline{B}01B^r$ et $B^l1\underline{0}B^r$ peuvent être représenté par un seul ruban

$$B^l(B11B)(00B1)(1BBB)B^r :$$

- le premier bloc de quatre lettres ($B11B$) représente un symbole B du premier ruban, le premier 1 du deuxième, la présence de la tête de lecture du premier sur cette case, et l'absence de la tête de lecture du deuxième sur cette case.
- le deuxième bloc de quatre lettres ($00B1$) représente le premier 0 du premier ruban, le premier 0 du deuxième, l'absence de la tête de lecture du premier sur cette case, et la présence de la tête de lecture du deuxième sur cette case.
- le troisième bloc de quatre lettres ($1BBB$) représente le premier 1 du premier ruban, un $B$ du deuxième, l'absence de la tête de lecture du premier sur cette case, et l'absence de la tête de lecture du deuxième sur cette case.
- et tous les autres blocs à gauche et à droite sont composés uniquement de symboles $B$.

Enfin, pour simuler une instruction de M2, (par exemple une instruction sur le premier ruban de M2), il faut parcourir le ruban de M1 jusqu'à trouver la position représentant la tête de lecture du premier ruban de M2, et effectuer sur le ruban les opérations correspondantes.

On voit qu'un tel encodage est relativement coûteux, puisqu'il nécessite de nombreux aller-retours sur le ruban de M1 pour simuler successivement des opérations sur le premier et sur le deuxième ruban de M2. L'exemple suivant illustre bien ce propos: le programme donné en exemple s'effectue en temps linéaire avec deux rubans. Avec un seul ruban, ré-écrire l'entrée à l'envers requiert un temps quadratique.

## 2.5 Exemple

La machine suivante, à deux rubans, ré-écrit son entrée à l'envers:

$$
\begin{array}{rll}
P = & & 1: \ right_1, \\
 & & 2: \ if_1\ B\ goto\ Parcours2\ else\ 3, \\
 & & 3: \ if_1\ 1\ goto\ P1Cas1\ else\ P1Cas0, \\
 & P1Cas1 & 4: \ write_2\ 1, \\
 & & 5: \ right_2, \\
 & & 6: \ goto\ 1, \\
 & P1Cas0 & 7: \ write_2\ 0, \\
 & & 8: \ right_2, \\
 & & 9: \ goto\ 1, \\
 & Parcours2 & 10: \ left_2, \\
 & & 11: \ if_2\ B\ goto\ Parcours3\ else\ 10, \\
 & Parcours3 & 12: \ right_2, \\
 & & 13: \ left_1, \\
 & & 14:: \ if_2\ B\ goto\ FIN\ else\ 15, \\
 & & 15: \ if_2\ 1\ goto\ P3Cas1\ else\ P3Cas0, \\
 & P3Cas1 & 16: \ write_1\ 1, \\
 & & 17: \ right_2, \\
 & & 18: \ goto\ 13, \\
 & P3Cas0 & 19: \ write_1\ 0, \\
 & & 20: \ right_2, \\
 & & 21: \ goto\ 13, \\
 & FIN & 22:
\end{array}
$$

Exécutons le programme $P$ sur l'entrée $x = 1.0$. On a alors $Readin(x) = (B^l\underline{B}10B^r, B^l\underline{B}B^r)$. La suite de transitions est

$$
\begin{array}{ccccccc}
(1, B^l\underline{B}10B^r, B^l\underline{B}B^r) & \to & (2, B^lB\underline{1}0B^r, B^l\underline{B}B^r) & \to & (3, B^lB\underline{1}0B^r, B^l\underline{B}B^r) & \to \\
(4, B^lB\underline{1}0B^r, B^l\underline{B}B^r) & \to & (5, B^lB\underline{1}0B^r, B^l\underline{1}B^r) & \to & (6, B^lB\underline{1}0B^r, B^l1\underline{B}B^r) & \to \\
(1, B^lB\underline{1}0B^r, B^l1\underline{B}B^r) & \to & (2, B^lB1\underline{0}B^r, B^l1\underline{B}B^r) & \to & (3, B^lB1\underline{0}B^r, B^l1\underline{B}B^r) & \to \\
(7, B^lB1\underline{0}B^r, B^l1\underline{B}B^r) & \to & (8, B^lB1\underline{0}B^r, B^l1\underline{0}B^r) & \to & (9, B^lB1\underline{0}B^r, B^l10\underline{B}B^r) & \to \\
(1, B^lB1\underline{0}B^r, B^l10\underline{B}B^r) & \to & (2, B^lB10\underline{B}B^r, B^l10\underline{B}B^r) & \to & (10, B^lB10\underline{B}B^r, B^l10\underline{B}B^r) & \to \\
(11, B^lB10\underline{B}B^r, B^l1\underline{0}B^r) & \to & (10, B^lB10\underline{B}B^r, B^l1\underline{0}B^r) & \to & (11, B^lB10\underline{B}B^r, B^l\underline{1}0B^r) & \to \\
(10, B^lB10\underline{B}B^r, B^l\underline{1}0B^r) & \to & (11, B^lB10\underline{B}B^r, B^l\underline{B}10B^r) & \to & (12, B^lB10\underline{B}B^r, B^l\underline{B}10B^r) & \to \\
(13, B^lB10\underline{B}B^r, B^lB\underline{1}0B^r) & \to & (14, B^lB1\underline{0}B^r, B^lB\underline{1}0B^r) & \to & (15, B^lB1\underline{0}B^r, B^lB\underline{1}0B^r) & \to \\
(16, B^lB1\underline{0}B^r, B^lB\underline{1}0B^r) & \to & (17, B^lB1\underline{1}B^r, B^lB\underline{1}0B^r) & \to & (18, B^lB1\underline{1}B^r, B^lB1\underline{0}B^r) & \to \\
(13, B^lB1\underline{1}B^r, B^lB1\underline{0}B^r) & \to & (14, B^lB\underline{1}1B^r, B^lB1\underline{0}B^r) & \to & (15, B^lB\underline{1}1B^r, B^lB1\underline{0}B^r) & \to \\
(19, B^lB\underline{1}1B^r, B^lB1\underline{0}B^r) & \to & (20, B^lB\underline{0}1B^r, B^lB1\underline{0}B^r) & \to & (21, B^lB\underline{0}1B^r, B^lB10\underline{B}B^r) & \to \\
(13, B^lB\underline{0}1B^r, B^lB10\underline{B}B^r) & \to & (14, B^l\underline{B}01B^r, B^lB10\underline{B}B^r) & \to & (22, B^l\underline{B}01B^r, B^lB10\underline{B}B^r) &
\end{array}
$$

L'instruction 22 étant vide, l'exécution s'arrête. Le résultat du calcul est alors $Readout(B^l\underline{B}01B^r, B^lB10\underline{B}B^r) = 0.1$.

# 3 Machine à Compteurs (MC)

On dispose d'un nombre fini, non borné, de compteurs (registres) $X_0, X_1, X_2, \cdots$ pour mémoriser des entiers naturels. Le langage d'instruction permet de tester si un compteur est à zéro, d'incrémenter/décrémenter un compteur de 1. La convention d'initialisation est que tous les compteurs sont à $\perp$, désignant la valeur "indéterminée", qui est interprétée pour le calcul comme un 0, sauf le premier compteur, contenant l'entrée. Le résultat est aussi donné par ce premier compteur lorsque l'exécution est finie.

## 3.1 Synthèse

- $MC - data = \mathbb{N}$
- $MC - store = \{\sigma | \sigma : \mathbb{N} \to \mathbb{N} \cup \{\bot\}\}$, où $\sigma(i)$ dénote le contenu du compteur $X_i$ pour $i \in \mathbb{N}$.
- $Readin(x) = \sigma : \begin{cases} 0 & \to x \\ i > 0 & \to \bot \end{cases}$
- $Readout(\sigma) = \sigma(0)$
- $MC - prog ::= X_i := X_i + 1 \mid X_i := X_i - 1 \mid if\ X_i = 0\ goto\ l'\ else\ l"$.

**Variantes** Dans le cas où on veut représenter une fonction qui prend deux entrées (par exemple, l'addition), la notation devient: $Readin(x_1, x_2) = \sigma : 0 \to x_1, 1 \to x_2$: les deux entiers $x_1$ et $x_2$ sont is dans les deux premiers registres $X_0$ et $X_1$. De la même manière, pour représenter une fonction qui retourne deux valeurs, la notation devient: $Readout(\sigma) = (\sigma(0), \sigma(1))$. On peut généraliser de la sorte pour n'importe quel nombre de valeurs à donner en entrée ou à lire en sortie.

## 3.2 Transitions

| | | | |
|---|---|---|---|
| $p \vdash (l, \sigma)$ | $\to$ | $\left(l+1, \sigma' : \begin{cases} i & \to \sigma(i)+1 \\ j \neq i & \to \sigma(j) \end{cases}\right)$ | si $I_l = X_i := X_i + 1$ |
| $p \vdash (l, \sigma)$ | $\to$ | $\left(l+1, \sigma' : \begin{cases} i & \to \sigma(i)-1 \\ j \neq i & \to \sigma(j) \end{cases}\right)$ | si $I_l = X_i := X_i - 1$ et $\sigma(i) \neq 0$ |
| $p \vdash (l, \sigma)$ | $\to$ | $(l+1, \sigma)$ | si $I_l = X_i := X_i - 1$ et $\sigma(i) = 0$ |
| $p \vdash (l, \sigma)$ | $\to$ | $(l', \sigma)$ | si $I_l = if\ X_i = 0\ goto\ l'\ else\ l"$ et $\sigma(i) = 0$ |
| $p \vdash (l, \sigma)$ | $\to$ | $(l", \sigma)$ | si $I_l = if\ X_i = 0\ goto\ l'\ else\ l"$ et $\sigma(i) \neq 0$ |

## 3.3 Exemple

La machine qui calcule le modulo à 2 de son entrée peut s'écrire comme suit:

$$
\begin{array}{rcll}
P & = & 1: & if\ X_0 = 0\ goto\ 7\ else\ 2, \\
 & & 2: & X_0 = X_0 - 1, \\
 & & 3: & if\ X_0 = 0\ goto\ 6\ else\ 4, \\
 & & 4: & X_0 = X_0 - 1, \\
 & & 5: & goto\ 1, \qquad\qquad (if\ X_0 = 0\ goto\ 1\ else\ 1) \\
 & & 6: & X_0 = X_0 + 1, \\
 & & 7: &
\end{array}
$$

Exécutons le programme $P$ sur l'entrée $x = 3$. On a alors $Readin(x) = \sigma : 0 \to 3$. La suite de transitions est

$$
\begin{array}{llllllllll}
(1, \sigma : 0 \to 3) & \to & (2, \sigma : 0 \to 3) & \to & (3, \sigma : 0 \to 2) & \to & (4, \sigma : 0 \to 2) & \to & (5, \sigma : 0 \to 1) & \to \\
(1, \sigma : 0 \to 1) & \to & (2, \sigma : 0 \to 1) & \to & (3, \sigma : 0 \to 0) & \to & (6, \sigma : 0 \to 0) & \to & (7, \sigma : 0 \to 1) & \to
\end{array}
$$

L'instruction 7 étant vide, l'exécution s'arrête. Le résultat du calcul est alors $Readout(\sigma) = \sigma(0) = 1$.

### 3.4 Extension de l'ensemble d'instructions

Les instructions suivantes, dont l'interprétation est évidente, sont facilement programmables avec les MC, et peuvent être utilisées dans l'écriture des programmes:

$$X_i := X_j \mid goto\ l \mid if\ X_i < X_j\ goto\ l'\ else\ l" \mid if\ X_i = X_j\ goto\ l'\ else\ l"$$
$$x_i := constante \mid if\ X_i = constante\ goto\ l'\ else\ l" \mid if\ X_i < constante\ goto\ l'\ else\ l"$$

Elles pourront également être utilisées pour les SRAM et les CRAM.

### 3.5 Machines de Turing et machines à compteurs

Les machines de Turing calculent des fonctions de $\{0,1\}^*$ vers $\{0,1\}^*$, les machines à compteurs des fonctions de $\mathbb{N}$ vers $\mathbb{N}$. Il est assez facile de mettre en bijection $\{0,1\}^*$ et $\mathbb{N}$: c'est précisément l'opération réalisée par l'encodage binaire des nombres entiers. Modulo cette bijection, les machines de Turing et les machines à compteurs calculent les mêmes fonctions.

Soit $f$ une fonction calculée par une machine à compteurs M1, utilisant $n$ compteurs. On peut aisément simuler le comportement de chaque compteur de M1 sur un ruban d'une machine de Turing à $n$ rubans: les opérations arithmétiques du langage des MC se calculent bien avec des MT (voir exercices de TD). De cette façon, il est facile de simuler une MC avec une MT.

Pour simuler une MT (à un ruban, pour simplifier) par une MC, il convient de représenter la mémoire de la MT avec des compteurs. On divise le ruban de la MT en deux parties, la partie à gauche de la tête de lecture, que l'on lit de gauche à droite depuis le premier symbole non blanc, et la partie à droite de la tête de lecture, que l'on lit de droite à gauche depuis le dernier symbole non blanc. Le sens de lecture ainsi spécifié permet d'associer à chaque demi-ruban un nombre entier: c'est l'entier obtenu par le décodage en base 3 (alphabet $\{0,1,B\}$ ) du mot lu. Les opérations sur le ruban de la MT se simulent alors facilement avec des opérations arithmétiques simples (modulo à 3, addition, soustraction, multiplication et division entière par 3).

## 4 Machine à adressage indirect (SRAM)

Ces machines sont une extension des machines abstraites à compteurs qui est plus proche du langage machine usuel. La mémoire est toujours formée de registres, mais on dispose d'un ensemble d'instructions plus riche. Les différentes définitions des RAM contiennent essentiellement la possibilité de:

- copier le contenu d'un registre dans un autre,
- faire de l'adressage indirect: lire la valeur d'un registre comme une "adresse mémoire" c'est à dire un numéro d'un autre registre, et accéder à cet autre registre (en lecture et/ou en écriture). C'est une abstraction des pointeurs, bien connus en programmation. Et enfin,
- faire des opérations élémentaires sur les valeurs des registres.

Dans ce modèle, Il n'y a pas de limite à la taille des mots ni à l'espace d'adresses mémoires : le nombre de registres est potentiellement infini et chacun peut contenir un entier naturel de taille arbitraire. Bien que l'exécution d'un programme ne puisse utiliser directement qu'un nombre fini de registres, ce nombre n'est pas fixé "en dur" dans le programme, mais peut dépendre de la valeur donnée en entrée à ce programme, et ainsi être arbitrairement grand.

## 4.1 Synthèse

Ce modèle diffère de la machine à compteurs (MC) uniquement par le jeu d'instructions

$$\begin{array}{rcl} SRAM-prog & ::= & X_i := X_i + 1 \mid X_i := X_i - 1 \mid if\ X_i = 0\ goto\ l'\ else\ l" \\ & \mid & X_i := X_j \mid X_i :=< X_j > \mid < X_i >:= X_j \end{array}$$

## 4.2 Transitions

| | | | |
|---|---|---|---|
| $p \vdash (l,\sigma)$ | $\rightarrow$ | $\left(l+1, \sigma' : \begin{cases} i & \rightarrow \sigma(i)+1 \\ j \neq i & \rightarrow \sigma(j) \end{cases}\right)$ | si $I_l = X_i := X_i + 1$ |
| $p \vdash (l,\sigma)$ | $\rightarrow$ | $\left(l+1, \sigma' : \begin{cases} i & \rightarrow \sigma(i)-1 \\ j \neq i & \rightarrow \sigma(j) \end{cases}\right)$ | si $I_l = X_i := X_i - 1$ et $\sigma(i) \neq 0$ |
| $p \vdash (l,\sigma)$ | $\rightarrow$ | $(l+1,\sigma)$ | si $I_l = X_i := X_i - 1$ et $\sigma(i) = 0$ |
| $p \vdash (l,\sigma)$ | $\rightarrow$ | $(l',\sigma)$ | si $I_l = if\ X_i = 0\ goto\ l'\ else\ l"$ et $\sigma(i) = 0$ |
| $p \vdash (l,\sigma)$ | $\rightarrow$ | $(l",\sigma)$ | si $I_l = if\ X_i = 0\ goto\ l'\ else\ l"$ et $\sigma(i) \neq 0$ |
| $p \vdash (l,\sigma)$ | $\rightarrow$ | $\left(l+1, \sigma' : \begin{cases} i & \rightarrow \sigma(\sigma(j)) \\ k \neq i & \rightarrow \sigma(k) \end{cases}\right)$ | si $I_l = X_i :=< X_j >$ |
| $p \vdash (l,\sigma)$ | $\rightarrow$ | $\left(l+1, \sigma' : \begin{cases} \sigma(i) & \rightarrow \sigma(j) \\ k \neq \sigma(i) & \rightarrow \sigma(k) \end{cases}\right)$ | si $I_l =< X_i >:= X_j$ |
| $p \vdash (l,\sigma)$ | $\rightarrow$ | $\left(l+1, \sigma' : \begin{cases} i & \rightarrow \sigma(j) \\ k \neq i & \rightarrow \sigma(k) \end{cases}\right)$ | si $I_l = X_i := X_j$ |

**Remarque** Etant donné une SRAM $P$, il est toujours possible d'écrire une SRAM $P'$ qui calcule exactement la même fonction que $P$, sans utiliser certains de ses registres. Par exemple, on peut s'assurer que $P'$ n'utilise aucun registre de numéro impair. Pour les opérations sur les registres en adressage direct, il suffit pour cela de multiplier par deux tous les indices des registres. Pour les opérations en adressage indirect, il faut multiplier par deux tous les valeurs utilisée comme adresse avant l'opération d'accès au registre, puis les re-diviser par deux après l'opération d'accès. On peut donc ainsi sans perte de généralité supposer que l'on dispose d'autant de jeux de registres que l'on souhaite, $X_i$, $i \in \mathbb{N}$, $Y_i$, $i \in \mathbb{N}$, etc, lorsque cela permet de faciliter l'écriture des programmes. L'exemple ci-dessous en est une illustration, qui utilise des noms de registres différents, $X_i$ et $R_i$.

## 4.3 Exemple

La machine qui vérifie si une liste d'entiers contient un 0 peut s'écrire comme suit. Initialement, $X_0$ contient la taille $n$ de la liste, et les registres $X_1, \cdots, X_n$ contiennent les valeurs des éléments de la liste. Les registres de travail sont notés $R_1 \cdots, R_k$, pour $k \in \mathbb{N}$.

La machine renvoie 1 (sur le registre $X_0$) si la liste $X_1, \cdots, X_n$ contient un 0, et 0 sinon.

$$\begin{array}{rcl} P & = & 1 : R_1 := X_0, \\ & & 2 : if\ R_1 = 0\ goto\ 7\ else\ 3, \\ & & 3 : R_2 :=< R_1 >, \\ & & 4 : if\ R_2 = 0\ goto\ 9\ else\ 5, \\ & & 5 : R_1 := R_1 - 1, \\ & & 6 : goto\ 2, \\ & & 7 : X_0 := 0, \\ & & 8 : goto\ 10, \\ & & 9 : X_0 := 1, \\ & & 10 : \end{array}$$

Exécutons le programme $P$ sur l'entrée $3, 7, 0, 8$. Pour faciliter la rédaction on note la fonction $\sigma$ par l'énumération explicite de ses valeurs: $Readin(x) = (X_0 : 3, X_1 : 7, X_2 : 0, X_3 : 8)$. La suite de transitions est

$$\begin{array}{ccccc}
(1, (X_0 : 3, X_1 : 7, X_2 : 0, X_3 : 8)) & \to & (2, (X_0 : 3, X_1 : 7, X_2 : 0, X_3 : 8, R_1 : 3)) & \to \\
(3, (X_0 : 3, X_1 : 7, X_2 : 0, X_3 : 8, R_1 : 3)) & \to & (4, (X_0 : 3, X_1 : 7, X_2 : 0, X_3 : 8, R_1 : 3, R_2 : 8)) & \to \\
(5, (X_0 : 3, X_1 : 7, X_2 : 0, X_3 : 8, R_1 : 3, R_2 : 8)) & \to & (6, (X_0 : 3, X_1 : 7, X_2 : 0, X_3 : 8, R_1 : 2, R_2 : 8)) & \to \\
(2, (X_0 : 3, X_1 : 7, X_2 : 0, X_3 : 8, R_1 : 3, R_2 : 8)) & \to & (3, (X_0 : 3, X_1 : 7, X_2 : 0, X_3 : 8, R_1 : 2, R_2 : 8)) & \to \\
(4, (X_0 : 3, X_1 : 7, X_2 : 0, X_3 : 8, R_1 : 3, R_2 : 0)) & \to & (9, (X_0 : 3, X_1 : 7, X_2 : 0, X_3 : 8, R_1 : 2, R_2 : 0)) & \to \\
(10, (X_0 : 1, X_1 : 7, X_2 : 0, X_3 : 8, R_1 : 3, R_2 : 0)) & & & &
\end{array}$$

L'instruction 10 étant vide, l'exécution s'arrête. Le résultat du calcul est alors $Readout(\sigma) = \sigma(0) = X_0 = 1$.

# 5 Machine abstraite parallèle (PRAM)

Il s'agit d'un modèle abstrait de calcul massivement parallèle pour le développement d'algorithmes. Une machine parallèle est constituée d'un nombre arbitrairement grand, non borné, de processeurs, qui effectuent des calculs en parallèle. Chaque processeur dispose d'une mémoire locale, isolée de celle des autres processeurs, et d'un PID (identifiant unique, qui lui est propre, et auquel il a accès). Cette mémoire locale est accessible par adressage direct ou indirect. En outre, la communication entre les différents processeurs est assurée par le biais d'un mémoire globale, partagée en lecture et en écriture entre tous les processeurs. Là encore, la mémoire partagée est accessible en adressage direct ou indirect.

On dispose d'un unique programme pour tous les processeurs, et tous les processeurs commencent le calcul à la première instruction. De plus, le modèle est synchrone : tous les processeurs passent d'une instruction à la suivante au même moment, les horloges sont synchronisées. L'exécution du même programme est différenciée pour chaque processeur grâce à la valeur de son PID: les branchements conditionnels liées à ce PID induiront des sauts différents pour chaque processeur.

Les modèles de PRAM sont basés sur les différents mode de lecture/écriture par plusieurs processeurs sur le même registre de la mémoire globale : EREW, CREW ou CRCW selon que la lecture (Read) ou l'écriture (Write) est Exclusive ou Concurrente. Dans le cas d'écriture concurrente en mémoire globale du modèle commun, on convient que tous les processeurs écrivent la même chose.

On détaille ici les CRAM. Une CRAM est une suite $\{R_i\}$, $i \geq 1$, de machines RAM. Chaque machine $R_k$ a sa propre mémoire locale: un ensemble infini de registres $X_i^k$, $i \in \mathbb{N}$, pouvant contenir un entier naturel. La mémoire globale un ensemble infini de registres $G_i$, $i \in \mathbb{N}$, accessibles simultanément en lecture/écriture par différents processeurs. En cas de tentative d'écriture simultanée d'une valeur différente par plusieurs processeurs, la résolution du conflit se fait par une gestion de priorité: seul le processeur de PID *le plus petit* réalise l'écriture. Les autres continuent leur exécution normalement, ils ne sont pas bloqués. De même, en cas d'opérations de lecture et d'écriture simultanée par deux processeurs, la valeur lue par le processeur lecteur est celle présente dans le registre *avant* l'opération d'écriture du deuxième processeur.

## 5.1 Synthèse

- $CRAM - data = \mathbb{N}$
- $CRAM - store = \{\sigma | \sigma : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \cup \{\perp\}\}$, où $\sigma(i, k)$ dénote le contenu du compteur $X_i^k$ pour $i \in \mathbb{N}$ et $k > 0$, et $\sigma(i, 0)$ dénote le contenu du compteur global $G_i$. (On rappelle qu'il n'y a pas de processeur de numéro 0 - les numéros commencent à 1. On peut donc utiliser cet indice 0 pour dénoter la mémoire globale)
- $Readin(x) = \sigma : \left\{ \begin{array}{ll} (0,0) & \to x \\ (i,k) \neq (0,0) & \to \perp \end{array} \right.$ (entrée dans le registre global $G_0$)
- $Readout(\sigma) = \sigma(0, 0)$ (valeur de retour dans le registre global $G_0$)

•

$$
\begin{array}{rcl}
CRAM-prog & ::= & X_i := X_i + 1 \mid X_i := X_i - 1 \mid if\ X_i = 0\ goto\ l'\ else\ l" \\
& \mid & X_i := X_j \mid X_i :=< X_j > \mid < X_i >:= X_j \\
& \mid & X_i := PID \mid X_i :=< X_j >^g \mid < X_i >^g := X_j \\
& \mid & X_i := G_j \mid G_i := X_i
\end{array}
$$

Lorsqu'on effectue une instruction sur un registre local $X_i$ pour une machine $R_k$ de la CRAM, la notation $X_i$ désigne le registre $X_i^k$.

**Remarque** Comme pour les modèles vus précédemment, les fonctions calculables par une CRAM sur une entrée finie sont exactement les fonctions calculables par MT, MC, ou par SRAM. La simulation d'une CRAM par une SRAM est néanmoins trop ardue pour être rapidement résumée. On peut se contenter ici de remarquer qu'un ordonanceur d'un système d'exploitation réalise exactement cette opération: simuler l'exécution parallèle de différents processus sur une machine séquentielle.

## 5.2 Transitions

Outre les transitions des SRAM, on a également les transitions suivantes pour chaque processeur:

| | | | |
|---|---|---|---|
| $p \vdash (l,\sigma)$ | $\rightarrow$ | $\left(l' : \begin{cases} k & \rightarrow l(k)+1 \\ t \neq k & \rightarrow l(t) \end{cases}, \sigma' : \begin{cases} (i,k) & \rightarrow k \\ (j,k), j \neq i & \rightarrow \sigma(j,k) \end{cases}\right)$ | si $I_{l(k)} = X_i := PID$ |
| $p \vdash (l,\sigma)$ | $\rightarrow$ | $\left(l' : \begin{cases} k & \rightarrow l(k)+1 \\ t \neq k & \rightarrow l(t) \end{cases}, \sigma' : \begin{cases} (i,k) & \rightarrow \sigma(\sigma(j,k),k) \\ (t,k), t \neq i & \rightarrow \sigma(t,k) \end{cases}\right)$ | si $I_{l(k)} = X_i :=< X_j >$ |
| $p \vdash (l,\sigma)$ | $\rightarrow$ | $\left(l' : \begin{cases} k & \rightarrow l(k)+1 \\ t \neq k & \rightarrow l(t) \end{cases}, \sigma' : \begin{cases} (i,k) & \rightarrow \sigma(\sigma(j,k),0) \\ (t,k), t \neq i & \rightarrow \sigma(t,k) \end{cases}\right)$ | si $I_{l(k)} = X_i :=< X_j >^g$ |
| $p \vdash (l,\sigma)$ | $\rightarrow$ | $\left(l' : \begin{cases} k & \rightarrow l(k)+1 \\ t \neq k & \rightarrow l(t) \end{cases}, \sigma' : \begin{cases} (\sigma(i,k),k) & \rightarrow \sigma(j,k) \\ (t,k), t \neq \sigma(i,k) & \rightarrow \sigma(t,k) \end{cases}\right)$ | si $I_{l(k)} =< X_i >:= X_j$ |
| $p \vdash (l,\sigma)$ | $\rightarrow$ | $\left(l' : \begin{cases} k & \rightarrow l(k)+1 \\ t \neq k & \rightarrow l(t) \end{cases}, \sigma' : \begin{cases} (\sigma(i,k),0) & \rightarrow \sigma(j,k) \\ (t,k), t \neq \sigma(i,k) & \rightarrow \sigma(t,k) \end{cases}\right)$ | si $I_{l(k)} =< X_i >^g := X_j$ |
| $p \vdash (l,\sigma)$ | $\rightarrow$ | $\left(l' : \begin{cases} k & \rightarrow l(k)+1 \\ t \neq k & \rightarrow l(t) \end{cases}, \sigma' : \begin{cases} (i,k) & \rightarrow \sigma(j,0) \\ (t,k), t \neq i & \rightarrow \sigma(t,k) \end{cases}\right)$ | si $I_{l(k)} = X_i := G_j$ |
| $p \vdash (l,\sigma)$ | $\rightarrow$ | $\left(l' : \begin{cases} k & \rightarrow l(k)+1 \\ t \neq k & \rightarrow l(t) \end{cases}, \sigma' : \begin{cases} (i,0) & \rightarrow \sigma(j,k) \\ (t,0), t \neq i & \rightarrow \sigma(t,0) \end{cases}\right)$ | si $I_{l(k)} = G_i := X_j$ |

Et on rappelle que une transition de la CRAM correspond à l'exécution au même moment d'une transition de chacun de ses processeurs, avec les règles de priorité précédemment données.

## 5.3 Exemple

La machine qui recherche si une valeur $e$ appartient à une liste d'entiers peut s'écrire comme suit. Initialement, la valeur à rechercher est dans le registre $G_0$, et les registres $G_1, \cdots, G_n$ contiennent les valeurs des éléments de la liste.

La machine renvoie 1 (sur le registre $G_0$) si la liste $G_1, \cdots, G_n$ contient la valeur recherchée, et 0 sinon.

$$
\begin{array}{rcl}
P & = & 1 : X_0 := G_0, \\
& & 2 : G_0 := 0, \\
& & 3 : X_1 := PID, \\
& & 4 : X_2 :=< X_1 >^g,
\end{array}
$$

$5 : if\ X_2 = X_0\ goto\ 6\ else\ 7,$
$6 : G_0 := 1,$
$7 :$

Exécutons le programme $P$ sur l'entrée $3, 1, 3, 5$. Pour faciliter la rédaction on note la fonction $\sigma$ par l'énumération explicite de ses valeurs: $Readin(x) = (G_0 : 3, G_1 : 1, G_2 : 2, G_3 : 5)$. On donne la suite de transitions pour 3 processeurs, sous forme de tableau, où une ligne correspond à un tic d'horloge, et une colonne correspond un un processeur.

| Registres $G_i$ | Proc. 1 | Proc. 2 | Proc. 3 |
|---|---|---|---|
| $(G_0 : 3, G_1 : 1, G_2 : 2, G_3 : 5)$ | $(1, )$ | $(1, )$ | $(1, )$ |
| $(G_0 : 3, G_1 : 1, G_2 : 2, G_3 : 5)$ | $(2, X_0 : 3)$ | $(2, X_0 : 3)$ | $(2, X_0 : 3)$ |
| $(G_0 : 0, G_1 : 1, G_2 : 2, G_3 : 5)$ | $(3, X_0 : 3)$ | $(3, X_0 : 3)$ | $(3, X_0 : 3)$ |
| $(G_0 : 0, G_1 : 1, G_2 : 2, G_3 : 5)$ | $(4, X_0 : 3, X_1 : 1)$ | $(4, X_0 : 3, X_1 : 2)$ | $(4, X_0 : 3, X_1 : 3)$ |
| $(G_0 : 0, G_1 : 1, G_2 : 2, G_3 : 5)$ | $(5, X_0 : 3, X_1 : 1, X_2 : 1)$ | $(5, X_0 : 3, X_1 : 2, X_2 : 3)$ | $(5, X_0 : 3, X_1 : 3, X_2 : 5)$ |
| $(G_0 : 0, G_1 : 1, G_2 : 2, G_3 : 5)$ | $(7, X_0 : 3, X_1 : 1, X_2 : 1)$ | $(6, X_0 : 3, X_1 : 2, X_2 : 3)$ | $(7, X_0 : 3, X_1 : 3, X_2 : 5)$ |
| $(G_0 : 1, G_1 : 1, G_2 : 2, G_3 : 5)$ | $(7, X_0 : 3, X_1 : 1, X_2 : 1)$ | $(7, X_0 : 3, X_1 : 2, X_2 : 3)$ | $(7, X_0 : 3, X_1 : 3, X_2 : 5)$ |

L'instruction 7 étant vide, l'exécution s'arrête là pour tous les processeurs. Le résultat du calcul est alors $Readout(\sigma) = \sigma(0, 0) = G_0 = 1$. On note que le numéro d'instruction exécuter par chaque processeur commence à différer à partir de la 6ème étape du calcul, et que cette divergence provient du fait que les processeurs on des PID différents.

# 6 Programmation par algèbre de fonctions

On change ici de présentation en calculant une fonction à l'aide d'une algèbre de fonctions $[\xi; OP]$. C'est à dire que l'on utilise des fonctions de base et des schémas pour définir d'autres fonctions : il faut les comprendre comme autant d'instructions de ce langage car chaque schéma de programmation exprime comment on construit une fonction, c'est un algorithme. Un exemple de fonction est la fonction constante zéro, notée 0, un exemple de schéma est la composition (Comp) à partir des fonctions $h, g_1, ..., g_m$ définie par

$$f(x_1, \cdots, x_n) = h(g_1(x_1, \cdots, x_n), \cdots, g_m(x_1, \cdots, x_n)).$$

On note alors $f = Comp(h, g_1, \cdots, g_m)$ et on dit que f est obtenu par composition à partir des fonctions $h, g_1, \cdots, g_m$.

Si on note $\mathcal{F}$ l'espace des fonctions, un opérateur est une fonctionelle $\mathcal{O} : \mathcal{F} \to \mathcal{F}$. Si $\xi$ est un ensemble de fonctions et OP est un ensemble d'opérateurs, alors $[\xi; OP]$ dénote le plus petit ensemble de fonctions contenant $\xi$ et clos sous les opérateurs de OP: c'est la clôture de $\xi$ par OP.

## 6.1 Fonctions Primitives Récursives

Les fonctions primitives récursives, de $\mathbb{N}^k \to \mathbb{N}$ (vues en cours de calculabilité) sont définies comme l'algèbre de fonctions $PR = [0, \Pi, s; Comp, PR]$, où :

- 0 est la fonction constante 0,
- $\Pi$ est l'ensemble de fonctions de projection $\pi_i^k$, où $\pi_i^k(x_1, \cdots, x_k) = x_i$,
- $s$ est la fonction successeur: $s(x) = x + 1$,
- $Comp$ est le schéma de composition évoqué plus haut, et

- $PR$ est le schéma de récursion primitive défini comme suit $f = PR(h, g)$ si et seulement si:

$$\begin{aligned} f(0, x_1, \cdots, x_n) &= g(x_1, \cdots, x_n) \\ f(s(x), x_1, \cdots, x_n) &= h(x, x_1, \cdots, x_n, f(x, x_1, \cdots, x_n)) \end{aligned}$$

### 6.1.1 Exemples

On définit l'addition comme suit:

Soit $g = \pi_1^1$, soit $h = Comp(s, \pi_3^3)$, l'addition est alors $add = PR(h, g)$. Autrement dit, on a:

$$\begin{aligned} add(0, y) &= y \qquad (= \pi_1^1(y)) \\ add(s(x), y) &= s(add(x, y)) \qquad (= s(\pi_3^3(x, y, add(x, y)))) \end{aligned}$$

De la même manière, on peut définir la multiplication:

$$\begin{aligned} mul(0, y) &= 0 \\ mul(s(x), y) &= add(y, mul(x, y)) \end{aligned}$$

Et l'exponentielle $y^x$:

$$\begin{aligned} exp(0, y) &= s(0) \\ exp(s(x), y) &= mul(y, exp(x, y)) \end{aligned}$$

## 6.2 Fonctions Primitives Récursives sur les Notations

On considère ici des fonctions sur les mots finis d'un alphabet fini - usuellement les mots booléens.

Pour un alphabet fini $A = \{a_1, \cdots, a_l\}$, on prend les fonctions de base suivantes:

- la fonction constante $\epsilon$ (le mot vide)
- $\Pi$, l'ensemble de fonctions de projection $\pi_i^k$, où $\pi_i^k(x_1, \cdots, x_k) = x_i$,
- $S$ l'ensemble des fonctions successeur sur les mots de l'alphabet: Pour $i = 1, \cdots, l$, $s_i(x) = a_i.x$, le mot obtenu en plaçant la lettre $a_i$ en tête du mot $x$.

Les fonctions récursives primitives sur les notations sont alors l'ensemble $PR_{not} = [\epsilon, \Pi, S; Comp, PR_{not}]$, où $PR_{not}$ est le schéma de récursion primitive sur les notations suivant:

$$\begin{aligned} f(\epsilon, x_1, \cdots, x_n) &= g(x_1, \cdots, x_n) \\ f(s_1(x), x_1, \cdots, x_n) &= h_1(x, x_1, \cdots, x_n, f(x, x_1, \cdots, x_n)) \\ &\vdots \\ f(s_l(x), x_1, \cdots, x_n) &= h_l(x, x_1, \cdots, x_n, f(x, x_1, \cdots, x_n)) \end{aligned}$$

Modulo le codage des entiers par des mots sur un alphabet fini, l'ensemble des fonctions primitives récursives et l'ensemble des fonctions primitives récursives sur les notations sont identiques.

## 6.3 Fonctions Récursives - Fonctions Récursives sur les Notations

L'ensemble des fonctions récursives est obtenu en enrichissant les opérateurs avec le schéma de *minimisation* suivant, qui renvoie le plus petit entier $y$ annulant $f$, s'il existe. S'il n'existe pas, la valeur renvoyée est $\perp$:

$$\mu(f)(x_1, \cdots, x_n) = \begin{cases} y & \text{si } f(y, x_1, \cdots, x_n) = 0 \text{ et } \forall z < y, f(z, x_1, \cdots, x_n) \neq 0 \\ \perp & \text{si } \forall y, f(y, x_1, \cdots, x_n) \neq 0 \end{cases}$$

La valeur de retour $\perp$ dénote un résultat indéterminé. Du point de vue de la sémantique d'exécution, $\perp$ dénote la non-terminaison du calcul. C'est donc une valeur absorbante pour le calcul: si une fonction a un de ses paramètres égal à $\perp$, son résultat est également $\perp$.

On note $REC : [0, \Pi, s; Comp, PR, \mu]$ l'ensemble des fonctions récursives.

Ce schéma est également applicable au cas des fonctions sur les mots d'un alphabet fini. En munissant l'alphabet $A = \{a_1, \cdots, a_l\}$ d'une relation d'ordre, et en prenant l'ordre lexicographique induit sur les mots finis de $A$, on obtient:

$$\mu_{not}(f)(x_1, \cdots, x_n) = \begin{cases} y & \text{si } f(y, x_1, \cdots, x_n) = \epsilon \text{ et } \forall z < y, f(z, x_1, \cdots, x_n) \neq \epsilon \\ \perp & \text{si } \forall y, f(y, x_1, \cdots, x_n) \neq \epsilon \end{cases}$$

L'ensemble des fonctions récursives sur les notations est alors $REC_{not} = [\epsilon, \Pi, S; Comp, PR_{not}, \mu_{not}]$. Ces deux ensembles de fonctions $REC$ et $REC_{not}$ sont là encore identiques, et vérifient la thèse de Church-Turing: ils coïncident avec l'ensemble des fonctions calculables par Machine de Turing, par Machines à Compteurs, Machines RAM, PRAM, etc.