Introduction à la calculabilité

Marc Chevalier

v0.1.1α : 17

22 janvier 2017

Résumé

Cet ouvrage a pour vocation de résumer de façon claire l'ensemble des résultats fondamentaux en calculabilité, en particulier en ce qui concerne la classe de puissance des machines de TURING. On s'attache à trouver ici une cohérence interne et des définitions et preuves formelles et rigoureuses. En outre, on trouve peu de sujets de recherches ou de résultats récents et spécialisés. Le but étant d'être aussi large et exhaustif que possible sur les modèles de bases. L'idéal poursuivit est un ouvrage auto-suffisant ne faisant appel à aucun résultat extérieur de calculabilité. Arrivé à la fin de l'ouvrage, le lecteur devrait avoir des compétences suffisantes en calculabilité pour pouvoir comprendre des concepts avancés et spécialisés.

En plus d'une préface sur les mathématiques sous-jacentes et ouvrage est constitué de cinq parties.

Dans un premier temps on définit les modèles de calculs usuels qu'on va étudier : les machines de TURING, les automates cellulaires, les fonctions récursives, le λ-calcul, les RAM, les machines à compteurs et les automates à piles multiples. Dans chaque cas on définit le modèle, explique le fonctionnement et, aussi souvent que possible, on donne un exemple de programme simulant le modèle.

Dans la seconde partie, on prouve que les sept modèles précédents sont équivalents.

On commence par prouver que les automates cellulaires, les fonctions récursives, le λ-calcul et les machines à deux piles sont immédiatement équivalent aux machines de TURING.

Par la suite, on montre l'équivalence des machines à plusieurs piles et des machines à plusieurs compteurs, ainsi que l'équivalence des machines à plusieurs compteurs et des machines à deux compteurs.

On prouve le reste des équivalence indirectement en montrant la supériorité de certains langages sur d'autres.

Dans la troisième partie, on donne des propriétés concernant la puissance de calcul de la classe de ces modèles. On aborde les notions de nombres et fonctions calculables et d'ensemble récursifs et récursivement énumérables.

Puis on s'intéresse aux machines de TURING à oracles pour donner un cadre formel à la comparaison de la difficulté de problèmes. On étudie rapidement la structure des classes de difficulté des problèmes en parlant de degrés de TURING.

On présente ensuite une hiérarchie des fonctions primitives récursives.

Enfin, on définit une hiérarchie de difficultés des problèmes basée sur l'expression des problèmes en tant que formule prénexe. On explicite également la correspondance qui existe entre cette hiérarchie et les degrés de TURING précédemment introduits.

Dans la quatrième partie, on liste plusieurs modèles TURING-complets qui ne sont pas des modèles de calculs intéressant pour la théorie de la calculabilité en soi mais dont la puissance peut être exploitée (ou amusante) dans d'autres domaines.

Aussi, on commencera par parler des langages de programmations et en quoi il est raisonnable de considérer qu'un ordinateur a la puissance de calcul d'une machine de TURING. On montrera un exemple d'automate cellulaire connu, le jeu de la vie de CONWAY, qui s'avère être, de façon surprenante, TURING-complet.

Enfin, on s'intéressera aux machines de TURING non déterministes en montrant que cela ne change pas la classe de calculabilité, ouvrant la voie à tout une famille de modèles non déterministes.

Enfin, dans la cinquième partie, on traite les modèles moins puissant. Le travail ici est bien moins exhaustif et consiste surtout à donner une idée de ce qui existe et de leurs puissances de calcul comparées.

Les modèles en questions sont les automates finis, les automate à pile et les automates linéairement bornés. On présente donc ensuite la hiérarchie de CHOMSKY qui organise tous ces modèles.

Pour finir, on expose un modèle qui est encore un sujet de recherche : les machines de TURING rouillées. Ce modèle a une puissance de calcul incertaine et qui ne rentre pas dans le cadre de la hiérarchie de CHOMSKY.

Sommaire

Préface concernant les notations et les considérations mathématiques

Mises en garde

L'union disjointe

On fera souvent des unions entre ensembles a priori sans relation, comme des alphabets ou des ensembles d'états. Il s'agit souvent d'union disjointe, bien que cela ne soit pas toujours explicité. L'union disjointe consiste en une union dans laquelle les éléments identiques provenant d'ensemble différents restent distinguables. On note la réunion disjointe de A et B :

$$A \sqcup B$$

En pratique, on la notera souvent comme la réunion usuelle. Formellement, on peut définir la réunion disjointe comme :

$$A \sqcup B = \{(a, \emptyset) \mid a \in A\} \cup \{(b, \{\emptyset\}) \mid b \in B\}$$

Cette définition a le mauvais goût de ne pas être commutative. On peut ainsi modifier la définition ou modifier le sens de l'égalité en ignorant la valeur propre du second élément des paires autrement que pour regrouper.

Le but de l'union disjointe, est que s'il existe un élément x tel que $x \in A$ et $x \in B$, on aura dans $A \sqcup B$, un élément x_A et un élément x_B correspondant respectivement au x de A et au x de B.

La propriété la plus remarquable est que

$$|A \sqcup B| = |A| + |B|$$

pour tout A et B. C'est une forme d'union qui permet de ne jamais perdre d'élément.

On ne donnera pas les détails qui sont des tracasseries d'ordre purement mathématique.

Décomposition des n-uplets

Cette notation est relativement peu formelle mais tout à fait claire néanmoins et sans risque d'ambiguïté.

Lors de la définition d'un type de n-uplet, comme les machines de TURING, les automates cellulaires, etc., on utilise des notations pour chaque élément de ce n-uplet.

Cette notation permettra d'accéder aux composantes selon leurs noms sous la forme d'une notation rappelant celle d'accès aux champs des structures en C, notamment.

Par exemple, si on définit un ensemble E des 2-uplets (n, f) où $n \in \mathbb{N}$ et $f \in \mathbb{R}^{\mathbb{R}}$ et qu'on prend un élément $p \in E$, alors $p.n$ désignera la première composante de la paire et $p.f$ désignera la seconde composante.

Cette notation est pratique car les notations choisies sont toujours adaptées à la nature de l'objet manipulé. Par exemple, δ représente toujours une fonction de transition, Q est toujours l'ensemble des états etc..

Notion de calculs équivalents

Les différents modèles de calculs n'ont pas la même façon de représenter les données. Aussi, il faut se poser la question de ce qui peut être considéré comme des résultats identiques.

Les principales structures de données sont les suivantes

— les compteurs (contenant un entiers) ou les entiers,
— les rubans et réseaux (indexés par $\mathbb{N}$, $\mathbb{Z}$ ou $\mathbb{Z}^n$) contenant des symboles d'un alphabet quelconque,
— les piles contenant des symboles d'un alphabet quelconque,
— les mémoires à accès arbitraires (indexées par $\mathbb{N}$) contenant des entiers,
— les arbres abstraits.

Voici une liste non exhaustive des équivalences supposées évidentes qui permettent de considérer que des structures différentes contiennent la même information.

— bijection entre deux alphabets,
— restriction d'un espace de type $\mathbb{Z}^n$ à un sous espace de type $a\mathbb{Z} + b$ où $(a, b) \in (\mathbb{Z}^n)^2$
— écriture ou lecture d'un nombre dans une base quelconque (en attribuant éventuellement une valeur bien choisie à chaque lettre d'un alphabet),
— équivalence entre une pile et une ruban indexé par $\mathbb{N}$,
— équivalence entre deux rubans indexés par $\mathbb{N}$ et un ruban indexé par $\mathbb{N}$.
— extraction d'une case d'une mémoire à accès arbitraire, d'un sous ensemble fini de cases ou d'un ensemble de la forme $a\mathbb{N} + b$,

— comptage du nombre de répétitions d'un motif dans un arbre (comme le nombre d'itérations dans $f(f(\ldots f(a)\ldots))$).

Ce ne sont que des transformations très locales qui ne calculent rien à proprement parler mais changent simplement le format de sortie.

Notations

On résume ici les notations utilisés par la suite.

TABLE 1 – Notations

Notation	Signification
F^E	Ensemble des fonctions de E dans F
$F^{(E)}$	Ensemble des fonctions de E dans F à support compact
$E \to F$	Fonction de E dans F
$E \hookrightarrow F$	Injection de E dans F
$E \twoheadrightarrow F$	Surjection de E dans F
$E \hookrightarrow\!\!\!\!\twoheadrightarrow F$	Bijection de E dans F
$\mapsto$	Image
$\circ$	Composition de fonctions
$E(F, G)$	Ensemble des fonctions de F dans G appartenant à E
$\prec$	Domination
$\perp$	Valeur non définie
$\forall$	Quantificateur universel
$\exists$	Quantificateur existentiel
$\wedge$	ET logique
$\vee$	OU logique
$\neg$	NON logique
$[P]$	Crochet de IVERSON
$\doteq$	Égalité de fonction ou de fonction calculée
$:=$	Définition d'égalité
$:\Leftrightarrow$	Définition d'équivalence
$[\![a, b]\!]$	Intervalle d'entier
$a \equiv b[k]$	Congruence de a et b modulo k
$\mathrm{Vect}_E(F)$	E-espace vectoriel engendré par F
$\|x\|_n$	Norme n
$\|x\|_\infty$	Norme infinie
$\mathbb{A}$	Ensemble des nombres algébriques
$E[X]$	Ensemble des polynômes sur E à une indéterminée
$E_n[X]$	Ensemble des polynômes de degré au plus n de $E[X]$
γ	Constante γ d'EULER-MASCHERONI
$\cup$	Réunion
$\sqcup$	Réunion disjointe
$\times$	Produit cartésien
$\in$	Appartenance
$\subseteq$	Inclusion

– Suite sur la page suivante –

TABLE 1 – Notations

Notation	Signification
$\subsetneq$	Inclusion stricte
$\setminus$	Différence ensembliste
$\mathcal{P}$	Ensemble des parties
$\lvert E \rvert$	Cardinal de E
$\emptyset$	Ensemble vide
$E^\star$	Fermeture de KLEENE de E
ε	Mot vide
$\lvert u \rvert$	Longueur de u
$\rightarrow^+$	Fermeture transitive de $\rightarrow$
$\rightarrow^*$	Fermeture réflexive et transitive de $\rightarrow$
$\leftrightarrow^*$	Fermeture réflexive, transitive et symétrique de $\rightarrow$
$\mathbb{M}$	Ensemble des machines de TURING
$\longleftarrow$	Mouvement gauche d'une machine de TURING
$\longrightarrow$	Mouvement droit d'une machine de TURING
`HALT`	Signal d'arrêt d'une machine de TURING
$\bigcirc$	opération neutre d'une machine de TURING
$\mathfrak{M}$	L'ensemble des machines de TURING rouillées
$\mathfrak{M}_f$	L'ensemble des machines de TURING rouillées au sens faible
$\mathfrak{C}(M, x)$	Nombre de transitions charnières de M sur x
$\mathfrak{C}(M)x$	Nombre d'oxydation de M
$\dot{-}$	Soustraction modifiée (soustraction dans $\mathbb{N}$)
$\mathcal{PR}$	Fonctions primitives récursives
$\mathcal{R}$	Fonctions récursives
$\mathrm{Rec}(g, h)$	Construction récursive à partir de g et h
$\mathrm{SumB}(f)$	Somme bornée de f
$\mathrm{ProdB}(f)$	Produit borné de f
$\mathrm{MinB}(f)$	Minimisation borne de f
$\mathrm{Min}(f)$	Minimisation de f
$\mathcal{A}$	Fonction d'ACKERMANN
H_n	$n^{\text{ème}}$ hyperopérateur
$\uparrow^n$	$n^{\text{ème}}$ flèche de KNUTH
$\mathfrak{f}_n$	$n^{\text{ème}}$ fonction de base de la hiérarchie de GRZEGORCZYK
π_1, π_2	Première et seconde projection pour les couples
$\mathcal{E}$	Classe des fonctions élémentaires
$\mathcal{E}^n$	Classe des fonctions de la hiérarchie de GRZEGORCZYK
$\leftrightarrow_\alpha$	α-équivalence
$\rightarrow_\beta$	β-réduction
$=_\beta$	β-équivalence

– Suite sur la page suivante –

TABLE 1 – Notations

Notation	**Signification**
$\leftrightarrow_\eta$	η-équivalence
$[M/x]\,e$	Substitution de x par M dans e
$\mathbb{T}$	Ensemble des nombres calculables
Σ_n Π_n Δ_n	Classes de la hiérarchie arithmétique
$\leqslant_T$	TURING-réductible
$\equiv_T$	TURING-équivalent
$\deg(A)$	Degré de TURING de A
X'	Saut de TURING de X
$X^{(n)}$	$n^{\text{ème}}$ saut de TURING de X
$\emptyset$	Degré de TURING de l'ensemble vide

Première partie

Les modèles de calculs

Chapitre 1

Les machines de TURING

1.1 Définition

Définition 1 (Machine de TURING [Car08a])**.**

Une machine de TURING est un 7-uplet $(Q, \Gamma, B, \Sigma, q_0, \delta, F)$ où
- Q est un ensemble fini d'états,
- Γ est l'alphabet de travail,
- $B \in \Gamma$ est un symbole particulier dit symbole blanc,
- Σ est l'alphabet des symboles en entrée ($\Sigma \subseteq \Gamma \setminus \{B\}$)
- $q_0 \in Q$ est l'état initial,
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\longleftarrow, \longrightarrow, \texttt{HALT}\}$ est la fonction de transition,
- $F \subseteq Q$ est l'ensemble des états acceptants.

Il existe un grand nombre d'autres formalismes pour les machines TURING, pas toujours rigoureux. Ils sont tous clairement équivalents. Ce formalisme particulier est extrêmement rare mais présente un certain nombre d'avantages. Entre autres, on distingue l'alphabet d'entrée et de travail, on précise le symbole blanc et l'état initial (formalisme insensible aux notations). D'autre part, ce modèle convient assez naturellement pour les problèmes de décision et de calcul.

Notation 1.

On note $\mathbb{M}$ l'ensemble des machines de TURING.

Au sens le plus stricte, il n'existe pas d'ensemble de toutes les machines de TURING. Mais il est inutile de les considérer toutes. En effet, des machines M_1 et M_2 peuvent être considérer identiques s'il existe deux bijections $\varphi : M_1.Q \to M_2.Q$ et $\psi : M_1.\Gamma \to M_2.\Gamma$ telles que :

— $\psi(M_1.B) = M_2.B$
— $\psi(M_1.\Sigma) = M_2.\Sigma$.
— $\varphi(M_1.q_0) = M_2.q_0$
— $\varphi(M_1.F) = M_2.F$
— $(q, a) \overset{M_1.\delta}{\to} (q', a', m) \Leftrightarrow (\varphi(q), \psi(a)) \overset{M_2.\delta}{\to} (\varphi(q'), \psi(a'), m)$

Si on quotiente par cette relation, ce qui sera systématiquement fait par la suite, le nombre de machine de TURING devient dénombrable.

1.2 Interprétation pragmatique

Intuitivement, une machine de TURING peut être vue comme une machine constituée d'un ruban, d'une tête de lecture/écriture, d'un registre d'état et d'un programme.

Le ruban, d'une longueur infinie, est divisé en cases consécutives chacune pouvant contenir un symbole d'un alphabet. Toute case contient par défaut un caractère spécial appelé caractère blanc. Le ruban peut être vu comme une liste indexée par $\mathbb{Z}$.

La tête de lecture correspond à une position sur le ruban. Celle-ci peut lire les caractères du ruban et ré-écrire dessus. Elle peut également se déplacer vers la droite ou la gauche d'une seule case à la fois.

Le registre d'état mémorise l'état actuel de la machine. Le nombre d'états possibles est toujours fini et il existe un état particulier appelé état de départ qui est l'état de la machine avant son exécution.

Le programme est une liste d'actions qui fait correspondre à tout couple constitué d'un symbole (celui qui est lu par la tête de lecture sur le ruban) et d'un état (l'état courant de la machine), un triplet constitué d'un symbole à écrire, d'un nouvel état à adopter et d'un mouvement de la tête de lecture ou d'un signal particulier appelé `HALT`qui arrête la machine. Si aucune action du programme ne correspond au couple en cours, la machine s'arrête.

Une machine de TURING est entièrement déterminée par sa configuration initiale, inscrite sur le ruban. Elle est alors dans son état initial.

Un algorithme s'exécute par étapes sur une machine de TURING. Lors d'une étape, la machine passe d'une configuration à la suivante grâce à la fonction de transition.

Avec le symbole situé sur la position de lecture et l'état en cours de la machine, elle détermine :

— le symbole qui sera écrit sur le ruban à la position actuelle,
— le déplacement de la position de lecture ($\longleftarrow$, $\longrightarrow$ ou `HALT`),
— le nouvel état de la machine.

Ainsi, une configuration d'une machine de TURING peut être décrite par différents triplets, notamment :

— le ruban, la position de lecture, l'état en cours,
— l'état de la machine, le ruban strictement à gauche de la position de lecture, le ruban à droite de la position de lecture.

Lorsque la machine reçoit le signal `HALT`, le calcul s'arrête. On dit que le calcul réussit si l'état lors de l'arrêt est un état acceptant. Sinon, le calcul échoue. On peut ainsi distinguer deux types d'arrêt. On s'en sert notamment afin de faire des tests dont le résultat est booléen. Comme l'appartenance d'un mot à un langage.

Il y a plusieurs manières d'interpréter le résultat d'une machine de TURING. Quand il s'agit d'un problème de décision, l'état final suffit. Sinon, on peut choisir que le ruban ne contient que le résultat du calcul, ou alors qu'il comporte un alphabet dédié au résultat. Le résultat peut aussi être juxtaposé à la suite de l'entrée. Une autre solution, qu'on utilisera par la suite à l'instar de TURING, est d'utiliser en alternance, des cases pour écrire le résultat et des cases pour le calcul intermédiaire.

Quand on utilise ce modèle, on peut imposer un ruban initialement vide ou contenant un mot initial fini. Dans tous les cas, un mot infini est interdit. Plus précisément, il ne doit y avoir un nombre cofini de symboles blancs. On impose très souvent une restriction sur la longueur des séquences de B qu'on peut trouver entre deux symboles initiaux. Souvent, on impose qu'il n'y ait pas de telles séquences, aussi les symboles initiaux forment une composante connexe, c'est un mot. Parfois, comme TURING le fait, on permet un certain nombre de B à chaque fois. Par exemple, TURING en autorisait un seul : deux B consécutif signifiait la fin de l'entrée.

1.3 Exemples

1.3.1 Exemple de problème de calcul

On va donner un exemple de machine qui incrémente un nombre donné en entrée sur son ruban. C'est un problème de calcul puisqu'il faut renvoyer une valeur et qu'il n'y a pas de tests dont on doit renvoyer la valeur.

On considère que le mot est donné en binaire sur le ruban. Aussi on utilise la machine suivante

— $Q = \{q_0\}$
— $\Gamma = \{0, 1, B\}$
— $\Sigma = \{0, 1\}$
— $F = \{q_0\}$

Et la table de transition suivante :

état / symbole	q_0
0	$q_0, 1, \texttt{HALT}$
1	$q_0, 0, \longrightarrow$
B	$q_0, 1, \texttt{HALT}$

On considère que le nombre est écrit dans le sens naturel et que la tête de lecture se trouve à l'origine sur le chiffre des unités. L'algorithme consiste à incrémenté le premier 0 rencontré en mettant les 1 rencontrés avant à 0.

On donne un exemple d'exécution sur l'entrée 19 :

q_0 : ... B B 1 0 0 1 1 B B ...
q_0 : ... B B 1 0 0 1 0 B B ...
q_0 : ... B B 1 0 0 0 0 B B ...
q_0 : ... B B 1 0 1 0 0 B B ...

On obtient ainsi bien l'écriture binaire du nombre 20.

On note qu'on n'a pas changé d'état durant le calcul : un seul et suffisant. Aussi, cet état est acceptant. Contrairement à d'autres formalismes de machines de TURING, être dans un état acceptant n'arrête pas la machine : cette distinction n'a d'importance qu'une fois la machine arrêtée.

Dans ce cas, il s'agit d'un problème de calcul, arrivé dans un état acceptant signifie juste que le calcul est terminé. C'est une caractérisation redondante avec l'arrêt de la machine mais cela permet de donner un modèle de machine de TURING qui unifie les problèmes de calcul et de décision.

1.3.2 Exemple de problème de décision

On va montrer comment reconnaitre les langages de DYCK avec une machine de TURING.

Intuitivement, les langages de DYCK sont les mots bien parenthésés. Aussi $()$ et $([]())$ appartiennent à des langages de DYCK, mais pas $([)]$.

Définition 2 (Langages de DYCK).

Soit A un alphabet fini, et soit $\overline{A} = \{\overline{a} \mid a \in A\}$ une copie de A disjointe de A. On note $\Sigma = A \sqcup \overline{A}$.
Soit $(u, v) \in (\Sigma^\star)^2$.

$$u \to v \Leftrightarrow \exists (x, y, a) \in (\Sigma^\star)^2 \times A : u = xa\overline{a}y \wedge v = xy$$

La réduction de DYCK est la fermeture transitive de cette relation.
Un mot de DYCK est un mot qui se réduit au mot vide ε.
Le langage de DYCK sur Σ est l'ensemble des mots de DYCK.

En effet, avec l'exemple précédent : $([]()) \to (()) \to () \to \varepsilon$

Le principe de l'algorithme de la machine de TURING qu'on va donner est de calculer de telles réductions. En pratique, le mot se raccourci en supprimant des lettres se trouvant au milieu. Sur un ruban, cela produit des décalages qui peuvent être difficilement gérables. On va utiliser une méthode courante consistant à marquer certains symboles pour être ignorés. Aussi la tête de lecture passera dessus mais l'état ne changera pas. On simule ainsi la suppression de ces lettres.

On suppose qu'on travaille avec un alphabet noté A. On note $\Sigma = A \sqcup \overline{A}$. Initialement, la tête de lecture est sur la première lettre du mot en entrée.

On travaille avec la machine :
— $Q = \{q_0, b, y\} \cup \{f_a\}_{a \in A} \cup \{e_a\}_{a \in A}$
— $\Gamma = \{B, I\} \cup \Sigma$
— $F = \{y\}$

La fonction de transition est décrite par :

symbole \ état	q_0	$f_s \quad s \in A$	$e_s \quad s \in A$	b
s			$b, I, \longleftarrow$	$b, s, \longleftarrow$
$a \in A$	$f_a, a, \longrightarrow$	$f_a, a, \longrightarrow$	$e_s, a,$ HALT	$b, a, \longleftarrow$
$\overline{s}$		$e_s, I, \longleftarrow$	$e_s, \overline{s},$ HALT	$b, \overline{s}, \longleftarrow$
$a \in \overline{A}$	$q_0, a,$ HALT	$f_s, a,$ HALT	$e_s, a,$ HALT	$b, a, \longleftarrow$
B	$y, B,$ HALT	$f_s, B,$ HALT	$e_s, B,$ HALT	$q_0, B, \longrightarrow$
I	$q_0, I, \longrightarrow$	$f_s, I, \longrightarrow$	$e_s, I, \longleftarrow$	$b, I, \longleftarrow$

On donne un exemple d'exécution de cette machine sur le mot $([]())$ pour $A = \{(, [\}$ et $\overline{A} = \{),]\}$.

TABLE 1.1 – Exemple d'exécution de la machine reconnaissant les langages de DYCK

q_0 :	...B	B	(	[	]	(	)	)	B B ...
$f_($:	...B	B	(	[	]	(	)	)	B B ...
$f_[$:	...B	B	(	[	]	(	)	)	B B ...
$e_[$:	...B	B	(	[	I	(	)	)	B B ...
b :	...B	B	(	I	I	(	)	)	B B ...
b :	...B	B	(	I	I	(	)	)	B B ...
q_0 :	...B	B	(	I	I	(	)	)	B B ...
$f_($:	...B	B	(	I	I	(	)	)	B B ...
$f_($:	...B	B	(	I	I	(	)	)	B B ...
$f_($:	...B	B	(	I	I	(	)	)	B B ...
$f_($:	...B	B	(	I	I	(	)	)	B B ...
$e_($:	...B	B	(	I	I	(	I	)	B B ...
b :	...B	B	(	I	I	I	I	)	B B ...
b :	...B	B	(	I	I	I	I	)	B B ...
b :	...B	B	(	I	I	I	I	)	B B ...
b :	...B	B	(	I	I	I	I	)	B B ...
q_0 :	...B	B	(	I	I	I	I	)	B B ...
$f_($:	...B	B	(	I	I	I	I	)	B B ...
$f_($:	...B	B	(	I	I	I	I	)	B B ...
$f_($:	...B	B	(	I	I	I	I	)	B B ...
$f_($:	...B	B	(	I	I	I	I	)	B B ...
$f_($:	...B	B	(	I	I	I	I	)	B B ...
$e_($:	...B	B	(	I	I	I	I	I	B B ...
$e_($:	...B	B	(	I	I	I	I	I	B B ...
$e_($:	...B	B	(	I	I	I	I	I	B B ...
$e_($:	...B	B	(	I	I	I	I	I	B B ...
$e_($:	...B	B	(	I	I	I	I	I	B B ...
b :	...B	B	I	I	I	I	I	I	B B ...
q_0 :	...B	B	I	I	I	I	I	I	B B ...
q_0 :	...B	B	I	I	I	I	I	I	B B ...
q_0 :	...B	B	I	I	I	I	I	I	B B ...
q_0 :	...B	B	I	I	I	I	I	I	B B ...
q_0 :	...B	B	I	I	I	I	I	I	B B ...

– Suite sur la page suivante –

TABLE 1.1 – Suite de l'exécution

q_0 :	...B	B	I	I	I	I	I	I	B B...
q_0 :	...B	B	I	I	I	I	I	I	B B...
q_0 :	...B	B	I	I	I	I	I	I	B B...

On voir clairement le fonctionnement de la machine. Celle-ci repasse régulièrement dans l'état q_0. Elle cherche une parenthèse ouvrante, en passant dans des états de la forme f_s. Lorsque la parenthèse fermante correspondante est rencontrée, on l'efface (on inscrit I) et on passe dans l'état e_s dont le but est d'effacer la parenthèse ouvrante correspondante. Si on ne rencontre pas la parenthèse correspondante (il n'y a plus de parenthèse fermante, la première parenthèse rencontrée est du mauvais type...) on refuse l'entrée. Une fois la parenthèse ouvrante effacé, on passe dans l'état b qui ramène au début du mot puis la machine reprend son travail dans l'état q_0. Elle réalise ainsi cette opération en boucle. Et chaque itération est une étape de la réduction de DYCK. Si le mot appartient au langage de DYCK, le mot finit par être complètement effacé. Alors, la machine dans l'état q_0 ne trouve plus de parenthèses et accepte l'entrée. Dans tous les autres cas, elle est refusée.

On peut aussi donner un exemple d'un mot non reconnu. On prend ici ([)].

q_0 :	...B	B	(	[	)	]	B B...
$f_($:	...B	B	(	[	)	]	B B...
$f_[$:	...B	B	(	[	)	]	B B...
$f_[$:	...B	B	(	[	)	]	B B...

1.4 Propriétés

Définition 3 (TURING-complétude).

Un système formel est TURING-complet s'il peut simuler toute machine de TURING.

Définition 4 (TURING-équivalence).

Un système formel qui calcule exactement le même ensemble de fonctions que les machines de TURING est dit TURING-équivalent.

Proposition 1.

Tout système TURING-équivalent est TURING-complet.

Définition 5 (Machine de TURING universelle).

Une machine de TURING est dite universelle si elle est capable de simuler toute machine de TURING.

On peut voir ça comme un interprète. Une telle machine prend sur son ruban la description de la machine à utiliser ainsi que le ruban de la machine à simuler et exécute la machine en question sur le ruban, en se servant éventuellement d'un espace de travail pour son exécution propre.

Proposition 2.

Toute machine de TURING universelle est TURING-équivalente.

On remarque qu'ici on ne dit rien sur l'existence de telles machines.

On se sert de cette propriété fortement. En effet, tout système dans lequel on peut simuler une machine de TURING universelle est TURING-complet.

Proposition 3.

Il existe des machines de TURING qui ne terminent pas.

Démonstration. Il suffit de prendre une machine qui fait un mouvement sans jamais de signal `HALT`. □

Théorème 1.

Il existe une machine de TURING universelle.

En effet, certaines machines de TURING universelles sont de notoriété publique.

Une machine de TURING universelle peut être facilement conçue par l'intuition. Si on appelle M la machine de TURING universelle et K la machine à simuler. On peut en effet coder la fonction de transition de K avec l'alphabet de M sur le ruban de M avant son lancement. Une portion du ruban contiendra donc la description de K alors que le reste pourra servir à l'exécution de K sur M.

Une telle machine sera explicitée dans la partie suivante, fournissant une preuve constructive.

Corollaire 1.

Il existe une infinité de machines de TURING universelles.

Démonstration. Il suffit de rajouter des états supplémentaires et inutiles à une machine de TURING universelle. □

1.5 Les machines de TURING universelles

Ici, le but est de prouver l'existence d'une machine de TURING universelle. Cette démontration est constructive. On explicitera la machine de TURING décrite dans *On computable numbers, with an Application to the Entscheidungsproblem* d'Alan TURING [Tur36]. Pour cela, on commence par donner le format et les restrictions utilisés sur ces machines. On introduit ensuite un formalisme bien utile : les m-fonctions. Enfin, on utilise ce formalisme pour définir des fonctions élémentaires pour machines de Turing qu'on assemblent ensuite pour obtenir la machine complète.

1.5.1 Les machines utilisées par TURING

Pour les besoins de sa démonstration, TURING a restreint l'ensemble des machines de TURING qu'il considérait. On peut prouver sans difficulté que l'ensemble des fonctions calculables par ces machines est identique à l'ensemble des fonctions calculables.

Ainsi on s'imposera les restrictions suivantes :

— On n'utilise que des 0 et des 1 pour inscrire les chiffres des séquences calculables.

— Le ruban n'est considéré infini que dans une direction (ruban indexé par $\mathbb{N}$). Les deux premières cases du ruban contiennent $\diamondsuit$.

— Les opérations licites sont l'inscription, le déplacement à gauche, à droite, l'effacement (équivalent à l'inscription du symbole blanc) et l'opération vide (représentée par $\bigcirc$) respectivement notés I, G, D, E, N.

— Le nombre d'opérations par transition n'est pas limité (inscription et déplacement).

TURING introduit la notion de E-case et de C-case. Les ruban est artificiellement divisé en une alternance de E-cases et de C-cases. Cette distinction est faite pour des raisons pratiques. Les C-cases sont utilisées pour stocker le résultat final. Tout symbole écrit dans une C-case pourra être consulté mais jamais modifié ou supprimé. Les E-cases servent à l'exécution de la machine simulée. Elles ont un intérêt particulier : permettre le marquage. On dit qu'une C-case est marquée par un caractère si ce caractère se trouve dans la E-case suivant la C-case.

$\diamondsuit$	$\diamondsuit$	C	α	A	$\cdot$	A	β

Ainsi dans cet exemple, le caractère C est marqué par α et le second A est marqué par β. Le premier A, quant à lui, n'est pas marqué.

On introduit la description d'une machine de la façon suivante. On se donne une machine $\mathcal{M}$ dont l'ensemble des symboles (aussi appelés m-configurations) est $\{S_0, \ldots, S_n\}$ et l'ensemble des états $\{q_0, \ldots, q_n\}$. On code une transition de la forme $\left(q_i, S_j\right) \mapsto \left(q_{i'}, S_{j'}, m\right)$ par le mot constitué de ; C suivi de i fois le caractère A, C, j fois B, C, j' fois B, puis D pour $m = \longrightarrow$, G pour $m = \longleftarrow$ ou N pour $m = \bigcirc$ suivi d'un C et de i' fois A. On obtient la description standard de $\mathcal{M}$ en concaténant les descriptions de toutes ses transitions. Ainsi une description standard est un mot de $\left(;CA^\star CB^\star CB^\star \left(D + G + N\right) CA^\star\right)^\star$. À partir de ce mot on peut définir un entier naturel appelé nombre descriptif en remplaçant les A par des 1, les B par des 2, les C par des 3, les D par des 4, les G par des 5, les N par des 6 et les ; par des 7.

On donne pour exemple une machine appelée $\mathcal{M}_{01}$ qui écrit dans les C-cases alternativement des 0 et des 1.

m-config.	symbole	opération	m-config. résultante
q_1	S_0	IS_1, D	q_2
q_2	S_0	IS_0, D	q_3
q_3	S_0	IS_2, D	q_4
q_4	S_0	IS_0, D	q_1

TABLE 1.2 – La table de transition de la machine $\mathcal{M}_{01}$

Elle peut se décrire sous la forme

$$;q_1S_0S_1Dq_2;q_2S_0S_0Dq_3;q_3S_0S_2Dq_4;q_4S_0S_0Dq_1$$

et admet donc comme description standard

```
;CACCBDCAA;CAACCDCAAA;CAAACCBBDCAAAA;CAAAACCDCA
```

TABLE 1.3 – Description standard de la machine $\mathcal{M}_{01}$

et comme nombre descriptif

73133253117311335311173111332253111173111133531

TABLE 1.4 – Nombre descriptif de la machine $\mathcal{M}_{01}$

Le nombre descriptif induit donc une injection de $\mathbb{M}$ dans $\mathbb{N}$.

1.5.2 Tables abrégées de TURING

La machine universelle réalise fréquemment certaines opérations comme la recherche, la copie ou la comparaison. On introduit pour écrire des tables abrégées, des m-configurations. Dans la suite, on note par des lettres majuscules des m-configurations et par des lettres minuscules grecques les symboles. Avec ce système la liste des configurations n'est pas explicitée. On peut utiliser le symbole # pour faire référence au symbole lu sur le ruban comme argument de m-configuration. On note $\rightarrow$ le passage de la machine dans un nouvel état. Un ensemble de tables abrégées codant une action (comme la recherche) est appelé fonction de m-configuration ou m-fonction.

Recherche

Cette première table permet d'expliquer sur un exemple le fonctionnement des tables abrégées.

m-config.	symbole	opérations	m-config. résultante
$f(\mathcal{E},\mathcal{B},\alpha)$	aucun	G	$f(\mathcal{E},\mathcal{B},\alpha)$
	$\diamondsuit$	G	$f_1(\mathcal{E},\mathcal{B},\alpha)$
	non $(\diamondsuit)$	G	$f(\mathcal{E},\mathcal{B},\alpha)$
$f_1(\mathcal{E},\mathcal{B},\alpha)$	aucun	D	$f_2(\mathcal{E},\mathcal{B},\alpha)$
	α	N	$\mathcal{E}$
	non(α)	D	$f_1(\mathcal{E},\mathcal{B},\alpha)$
$f_2(\mathcal{E},\mathcal{B},\alpha)$	aucun	D	$\mathcal{B}$
	α	N	$\mathcal{E}$
	non(α)	D	$f_1(\mathcal{E},\mathcal{B},\alpha)$

TABLE 1.5 – f [find] : $f(\mathcal{E},\mathcal{B},\alpha)$ trouve le premier α du ruban et $\rightarrow \mathcal{E}$ ou $\rightarrow \mathcal{B}$ s'il n'y a pas de α

Il est important de noter que $f(\mathcal{E},\mathcal{B},\alpha)$ ne représente pas une fonction f appliquée à trois arguments : $\mathcal{E},\mathcal{B}$ et α. Cette écriture correspond à une m-configuration. Les arguments $\mathcal{E},\mathcal{B}$ et α caractérisent le symbole recherché et les m-configurations de sortie possibles.

Pour chaque $(E,B,a) \in Q^2 \times \Gamma$, on peut réécrire la table de façon explicite en considérant que $f(E,B,a)$, $f_1(E,B,a)$ et $f_2(E,B,a)$ sont trois états de la machine. On obtient ainsi la liste explicite des m-configurations d'une telle machine en substituant de toutes les façon possibles les arguments de la m-fonction.

Pour rechercher le caractère α on met la machine dans la m-configuration $f(\mathcal{E}, \mathcal{B}, \alpha)$. Tant que la machine ne tombe pas sur $\diamondsuit$, elle reste dans cet état en se déplaçant à gauche. Elle revient ainsi au début du ruban. Elle passe ensuite dans l'état $f_1(\mathcal{E}, \mathcal{B}, \alpha)$. Puis la machine va à droite jusqu'à trouver α. Si la machine trouve une case vide, elle passe dans l'état $f_2(\mathcal{E}, \mathcal{B}, \alpha)$. S'il y a de nouveau une case vide, la machine passe dans l'état $\mathcal{B}$. Ainsi, si la machine trouve 2 cases vides consécutives, la m-fonction considère la suite du ruban est vide.

Suppression

On peut aussi montrer un exemple plus complexe.

m-config.	symbole	opérations	m-config. résultante
$e(\mathcal{E}, \mathcal{B}, \alpha)$			$f(e_1(\mathcal{E}, \mathcal{B}, \alpha), \mathcal{B}, \alpha)$
$e_1(\mathcal{E}, \mathcal{B}, \alpha)$		E	$\mathcal{E}$
$e(\mathcal{B}, \alpha)$			$e(e(\mathcal{B}, \alpha), \mathcal{B}, \alpha)$

TABLE 1.6 – e [erase] : $e(\mathcal{E}, \mathcal{B}, \alpha)$ efface le premier α du ruban et $\rightarrow \mathcal{E}$ ou $\rightarrow \mathcal{B}$ s'il n'y a pas de α sur le ruban. $e(\mathcal{B}, \alpha)$ efface tous les α et $\rightarrow \mathcal{B}$.

On note aussi que deux m-fonctions peuvent se différencier par le nom mais aussi par le nombre d'arguments (aussi appelé arité). Aussi $e(\mathcal{E}, \mathcal{B}, \alpha)$ et $e(\mathcal{B}, \alpha)$ sont deux m-fonctions différentes.

Ajout à la fin

m-config.		symbole	opérations	m-config. résultante
$pe(\mathcal{E}, \beta)$				$f(pe_1(\mathcal{E}, \beta), \mathcal{E}, \diamondsuit)$
$pe_1(\mathcal{E}, \beta)$	$\left\{\right.$	aucun	$I\beta$	$\mathcal{E}$
	$\left.\right\}$	quelconque	D, D	$pe_1(\mathcal{E}, \beta)$

TABLE 1.7 – pe [print at end] : $pe(\mathcal{E}, \beta)$ écrit β à la fin du ruban et $\rightarrow \mathcal{E}$

On peut généraliser la fonction pe en pe_n qui écrit une suite finie de symbole à la fin du ruban avant de changer d'état en posant $pe_n(\mathcal{E}, \beta_n, \ldots, \beta_1) = pe(pe_{n-1}(\mathcal{E}, \beta_2, \ldots, \beta_n), \beta_1)$.

Recherche et mouvement

m-config.	symbole	opérations	m-config résultante
$l(\mathcal{E})$		G	$\mathcal{E}$
$r(\mathcal{E})$		D	$\mathcal{E}$
$f'(\mathcal{E}, \mathcal{B}, \alpha)$			$f(l(\mathcal{E}), \mathcal{B}, \alpha)$
$f''(\mathcal{E}, \mathcal{B}, \alpha)$			$f(r(\mathcal{E}), \mathcal{B}, \alpha)$

TABLE 1.8 – f' : la machine s'arrête à gauche du premier α trouvé et $\rightarrow \mathcal{E}$. S'il n'y a pas de α, la machine $\rightarrow \mathcal{B}$
f'' : la machine s'arrête à droite du premier α trouvé et $\rightarrow \mathcal{E}$. S'il n'y a pas de α, la machine $\rightarrow \mathcal{B}$

Copie

m-config.	symbole	opérations	m-config résultante
$c(\mathcal{E}, \mathcal{B}, \alpha)$			$f'(c_1(\mathcal{E}), \mathcal{B}, \alpha)$
$c_1(\mathcal{E})$			$pe(\mathcal{E}, \#)$

TABLE 1.9 – c [copy] : $c(\mathcal{E}, \mathcal{B}, \alpha)$ copie le premier symbole marqué par α à la fin du ruban et $\rightarrow \mathcal{E}$. Si aucun symbole n'est marqué par α, la machine $\rightarrow \mathcal{B}$

On rappelle que # désigne le symbole sous la tête de lecture. Cette notation évite d'avoir à ré-écrire la transition pour chaque symbole possible.

Copie et suppression

m-config.	symbole	opérations	m-config. résultante
$ce(\mathcal{E}, \mathcal{B}, \alpha)$			$c(e(\mathcal{E}, \mathcal{B}, \alpha), \mathcal{B}, \alpha)$
$ce(\mathcal{B}, \alpha)$			$ce(ce(\mathcal{B}, \alpha), \mathcal{B}, \alpha)$

TABLE 1.10 – ce [copy and erase] : $ce(\mathcal{E}, \mathcal{B}, \alpha)$ copie le premier symbole marqué avec α à la fin du ruban et supprime la marque

$ce(\mathcal{B}, \alpha)$ copie dans l'ordre à la fin du ruban les symboles marqués avec α et supprime toutes les marques.

On généralise : $ce_n(\mathcal{B}, \alpha_1, \ldots, \alpha_n) := ce(ce_{n-1}(\mathcal{B}, \alpha_2, \ldots, \alpha_n), \alpha_1)$. Cette m-fonction copie dans l'ordre les symboles marqués avec α_1 puis avec α_2 ...

Substitution

m-config.	symbole	opérations	m-config résultante
$re\,(\mathcal{E},\mathcal{B},\alpha,\beta)$			$f\,(re_1\,(\mathcal{E},\mathcal{B},\alpha,\beta)\,,\mathcal{B},\alpha)$
$re_1\,(\mathcal{E},\mathcal{B},\alpha,\beta)$		$E, I\beta$	$\mathcal{E}$
$re\,(\mathcal{B},\alpha,\beta)$			$re\,(re\,(\mathcal{B},\alpha,\beta)\,,\mathcal{B},\alpha,\beta)$

TABLE 1.11 – *re* [replace] : $re\,(\mathcal{E},\mathcal{B},\alpha,\beta)$ remplace le premier α par β et $\rightarrow \mathcal{E}$. S'il n'y a pas de α, la machine $\rightarrow \mathcal{B}$
$re\,(\mathcal{B},\alpha,\beta)$ remplace tous les α par β et $\rightarrow \mathcal{B}$

Copie d'une séquence

m-config.	symbole	opérations	m-config résultante
$cr\,(\mathcal{E},\mathcal{B},\alpha)$			$c\,(re\,(\mathcal{E},\mathcal{B},\alpha,\mathtt{a})\,,\mathcal{B},\alpha)$
$cr\,(\mathcal{B},\alpha)$			$cr\,(cr\,(\mathcal{B},\alpha)\,,re\,(\mathcal{B},\mathtt{a},\alpha)\,,\alpha)$

TABLE 1.12 – $cr\,(\mathcal{B},\alpha)$ copie des symboles marqués avec a à la fin du ruban et $\rightarrow \mathcal{B}$

Comparaison

m-config.	symbole	op.	m-config résultante
$cp\,(\mathcal{E}_1,\mathcal{U},\mathcal{E},\alpha,\beta)$			$f'\,(cp_1\,(\mathcal{E}_1,\mathcal{U},\beta)\,,f\,(\mathcal{U},\mathcal{E},\beta)\,,\alpha)$
$cp_1\,(\mathcal{E},\mathcal{U},\beta)$			$f'\,(cp_2\,(\mathcal{E},\mathcal{U},\#)\,,\mathcal{U},\beta)$
$cp_2\,(\mathcal{E},\mathcal{U},\gamma)$	γ	$I\beta$	$\mathcal{E}$
	$\mathrm{non}(\gamma)$	D, D	$\mathcal{U}$
$cpe\,(\mathcal{E}_1,\mathcal{U},\mathcal{E},\alpha,\beta)$			$cp\,(e\,(e\,(\mathcal{E}_1,\mathcal{E},\beta)\,,\mathcal{E},\alpha)\,,\mathcal{U},\mathcal{E},\alpha,\beta)$
$cpe\,(\mathcal{U},\mathcal{E},\alpha,\beta)$			$cpe\,(cpe\,(\mathcal{U},\mathcal{E},\alpha,\beta)\,,\mathcal{U},\mathcal{E},\alpha,\beta)$

TABLE 1.13 – *cp* [compare] : $cp\,(\mathcal{E}_1,\mathcal{U},\mathcal{E},\alpha,\beta)$ compare les premiers caractères marqués avec α et β puis $\rightarrow \mathcal{E}$ s'il n'y a ni α ni β, $\rightarrow \mathcal{E}_1$ si ces caractères sont identiques et $\rightarrow \mathcal{U}$ dans tous les autres cas.
$cpe\,(\mathcal{E}_1,\mathcal{U},\mathcal{E},\alpha,\beta)$ est similaire à $cp\,(\mathcal{E}_1,\mathcal{U},\mathcal{E},\alpha,\beta)$ mais supprime les premiers α et β.
$cpe\,(\mathcal{U},\mathcal{E},\alpha,\beta)$ compare les séquences marquées avec α et β. Si ces séquences sont identiques $\rightarrow \mathcal{E}$, $\rightarrow \mathcal{U}$ sinon.

Recherche de la dernière occurrence

m-config.	symbole	opérations	m-config résultante
$q(\mathcal{E})$	aucun	D	$q(\mathcal{E})$
	quelconque	D	$q_1(\mathcal{E})$
$q_1(\mathcal{E})$	aucun	D	$q(\mathcal{E})$
	quelconque		$\mathcal{E}$
$q(\mathcal{E}, \alpha)$			$q(q_1(\mathcal{E}, \alpha))$
$q_1(\mathcal{E}, \alpha)$	α	D	$\mathcal{E}$
	non(α)	G	$q_1(\mathcal{E}, \alpha)$

TABLE 1.14 – $q(\mathcal{E}, \alpha)$ trouve le dernier α et $\rightarrow \mathcal{E}$

1.5.3 La machine universelle selon TURING

On note $\mathcal{U}$ la machine universelle et $\mathcal{M}$ la machine simulée

On s'impose quelques conditions de travail concernant la machine de TURING universelle et la machine simulée. Les conditions suivantes restreignent évidemment l'ensemble des machines simulables mais pas l'ensemble des fonctions calculables.

— Le ruban de $\mathcal{U}$ contient $\diamondsuit\diamondsuit$ (une C-case et une E-case), suivi de la description standard de $\mathcal{M}$ sur les C-cases et de ‡ dans la C-case suivant immédiatement la description standard de $\mathcal{M}$. Les E-cases sont initialement vides.
— $\mathcal{U}$ peut écrire les symboles `A`, `B`, `C`, `0`, `1`, `u`, `v`, `w`, `x`, `y`, `z` et `:`.
— Le reste du ruban est initialement vide (symbole `_`).
— L'état initial de $\mathcal{U}$ est b.

La machine universelle a les m-configurations suivantes :

m-config.	symbole	opérations	m-config résultante
$e(anf)$	$\diamondsuit$	D	$e_1(anf)$
	non($\diamondsuit$)	G	$e(anf)$
$e_1(anf)$	aucun		anf
	quelconque	D,E,D	$e_1(anf)$

La machine commence par revenir au premier $\diamondsuit$ qui est situé sur une E-case puis efface les E-cases jusqu'à tomber sur une E-case vide.

m-config.	symbole	opérations	m-config résultante
$con\,(\mathcal{E},\alpha)$	non(A)	D,D	$con\,(\mathcal{E},\alpha)$
	A	$G, I\alpha, D$	$con_1\,(\mathcal{E},\alpha)$
$con_1\,(\mathcal{E},\alpha)$	A	$D, I\alpha, D$	$con_1\,(\mathcal{E},\alpha)$
	C	$D, I\alpha, D$	$con_2\,(\mathcal{E},\alpha)$
	aucun	$IC, D, I\alpha, D, D, D$	$\mathcal{E}$
$con_2\,(\mathcal{E},\alpha)$	B	$D, I\alpha, D$	$con_2\,(\mathcal{E},\alpha)$
	non(B)	D,D	$\mathcal{E}$

Une m-fonction supplémentaire est indispensable pour le fonctionnement de $\mathcal{U}$. Son interprétation ne se comprend que dans le cas du formalisme imposé. Il sert à la recherche de la transition.

Table de $\mathcal{U}$

m-config.	symbole	opérations	m-config résultante
b			$f\,(b_1, b_1, \ddagger)$
b_1		$D, D, I\,(:\_C\_A)$	anf
anf			$q\,(anf_1, :)$
anf_1			$con\,(fom, y)$
fom	;	D, Iz, G	$con\,(fmp, x)$
	z	G, G	fom
	non$(;, z)$	G	fom
fmp			$cpe\,(e\,(e\,(anf, y)\,, x)\,, sim, x, y)$

La machine admet b comme état initial. On a d'abord la phase d'initialisation. Dans un premier temps, $\mathcal{U}$ cherche le symbole ‡ marquant la fin de la description standard de $\mathcal{M}$ et $\rightarrow b_1$. Dans l'état b_1, $\mathcal{M}$ écrit : CA sur les C-cases suivantes puis $\rightarrow anf$. C'est la fin de la phase d'initialisation.

Les états anf, anf_1, fom et fmp cherchent la transition dans la description standard correspondant à l'état actuel de $\mathcal{M}$ et au symbole lu. Pour ce faire, $\mathcal{U}$ marque avec z les ; des transitions déjà examinées. À l'issue de cette phase, la première transition marquée avec z est donc nécessairement celle qui convient. Si aucune transition ne convient, $\mathcal{U}$ lit des cases à gauche des $\diamondsuit$, ce qui se traduit par un arrêt de $\mathcal{U}$.

On a alors identifié la transition à appliquer. On passe alors à la phase de simulation.

m-config.		symbole	opérations	m-config résultante
sim				$f'\,(sim_1, sim_1, z)$
sim_1				$con\,(sim_2,)$
sim_2	$\left\{\right.$	A		sim_3
		non(A)	G, Iu, D, D, D	sim_2
sim_3	$\left\{\right.$	non(A)	G, Iy	$e\,(mf, z)$
		A	G, Iy, D, D, D	sim_3

On marque avec u le mot codant le symbole que $\mathcal{M}$ doit écrire et le symbole codant le mouvement à effectuer. De plus, on note avec y le nouvel état de $\mathcal{M}$.

Puis on efface les marques z.

On passe dans l'état mf. $\mathcal{U}$ se trouve alors au bout du ruban déjà parcouru.

m-config.		symbole	opérations	m-config résultante
mf				$q\,(mf_1, :)$
mf_1	$\left\{\right.$	A	G, G, G, G	mf_2
		non(A)	D, D	mf_1
mf_2	$\left\{\right.$	B	D, Ix, G, G, G	mf_2
		:		mf_4
		C	D, Ix, G, G, G	mf_3
mf_3	$\left\{\right.$	non$(:)$	D, Iv, G, G, G	mf_3
		:	G, Iu, D, D, D	mf_4
mf_4				$con\,(l\,(l\,(mf_5))\,, \_)$
mf_5	$\left\{\right.$	quelconque	D, Iw, D	mf_5
		aucun	I :	sh

m-config.		symbole	opérations	m-config. résultante
sh				$f\,(sh_1, inst, u)$
sh_1			G, G, G	sh_2
sh_2	$\left\{\right.$	C	D, D, D, D	sh_3
		non(C)		$inst$
sh_3	$\left\{\right.$	B	D, D	sh_4
		non(B)		$inst$
sh_4	$\left\{\right.$	B	D, D	sh_5
		non(B)		$pe_2\,(inst, 0, :)$
sh_5	$\left\{\right.$	B		$inst$
		non(B)		$pe_2\,(inst, 1, :)$

Ces m-fonctions permettent d'écrire sur la première C-case libre le symbole de $\mathcal{M}$ à écrire. Conformément aux hypothèses, ce symbole est 0 ou 1. On inscrit ensuite dans la C-case suivante ":".

$\mathcal{U}$ passe dans l'état *inst* et se trouve sur le dernier symbole inscrit.

m-config.	symbole	opérations	m-config résultante
$inst$			$q\left(l\left(inst_1\right), u\right)$
$inst_1$		D, E	$inst_1\left(\#\right)$
$inst_1\left(G\right)$			$ce_5\left(ov, v, y, x, u, w\right)$
$inst_1\left(D\right)$			$ce_5\left(ov, v, x, u, y, w\right)$
$inst_1\left(N\right)$			$ce_5\left(ov, v, x, y, u, w\right)$
ov			$e\left(anf\right)$

Ici, $\mathcal{U}$ cherche le dernier symbole marqué par u. La marque est effacée et les données concernant l'exécution de $\mathcal{M}$ sont copiées à la fin. Les marques sont effacées.

$\mathcal{U}$ passe dans l'état anf qui est l'état qui suit immédiatement la phase d'initialisation. On se retrouve ainsi dans une configuration semblable à la configuration à l'issue de la phase d'initialisation.

Le ruban a la structure suivante :

— $\diamondsuit\diamondsuit$ (sur une C-case puis une E-case).
— La description standard de $\mathcal{M}$ sur les C-cases.
— ‡ sur la C-case suivante.
— Une succession de : suivie des configurations prises par $\mathcal{M}$ puis de :, d'un 0 ou d'un 1 et enfin d'un :.

Le résultat de l'exécution de $\mathcal{M}$ est dans des C-cases. Pour obtenir le résultat, il suffit de récupérer dans l'ordre les 0 et les 1.

1.6 Simulation en C++

Cette section est dédiée à la simulation des machines de TURING utilisant des fonctions de m-configuration et des m-transitions classiques.

Le but ultime est de simuler la machine de TURING universelle décrite précédemment, observer et confirmer son bon fonctionnement. C'est grâce à ce programme que les erreurs ont été identifiées et corrigées.

Le programme est entièrement écris en C++ et ne comprend pas d'interface d'aucune sorte, ni d'acquisition de données depuis des fichiers. Il serait aisé de les ajouter mais cela dépasse le cadre stricte de la simulation.

`fonction.h`

```
#ifndef FONCTION_H_INCLUDED
#define FONCTION_H_INCLUDED

#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <cstdio>
#include <cstdlib>
#include <fstream>

void print(const std::vector<std::string>& e);
void print(const std::vector<std::pair<std::string,std::string> >& e);
void print(const std::vector<std::string>& e, std::ofstream& save);
void print(const std::vector<std::pair<std::string,std::string> >& e,
    std::ofstream& save) __attribute__((pure)) ;
bool is_m_function(const std::string& m_conf) __attribute__((pure));
unsigned int arite_m_conf(const std::string& m_conf) __attribute__((pure));
std::string name_m_conf(const std::string& m_conf);
std::vector<std::string> analyse_syntax_m_conf(const std::string& m_conf);
std::vector<std::string> analyse_syntax_symbole(const std::string& symbole);
bool analyse_lex_symbole(const std::string& filtrage,
    const std::string& symbole);
std::vector<std::pair<std::string,std::string> > analyse_syntax_operation(
    const std::string& operations, const std::vector<std::string>& m_conf,
    const std::vector<std::string>& m_fonc);
std::string substitution(const std::vector<std::string>& m_fonc,
    const std::vector<std::string>& m_conf,
    const std::vector<std::string>& result, const std::string& sym);
std::string substitution_symbole(const std::vector<std::string>& m_fonc,
    const std::vector<std::string>& m_conf,
    const std::vector<std::string>& sym);

#endif // FONCTION_H_INCLUDED
```

`fonction.cpp` (fonctions d'affichage et de sauvegarde)

```cpp
#include "fonction.h"
#include<iostream>

using namespace std;

void print(const vector<string>& e)
{
    cout<<"$";
    for(unsigned int i=0;i<e.size();++i)
        cout<<e[i]<<"|";
    cout<<"$"<<endl;
}

void print(const vector<pair<string,string> >& e)
{
    cout<<"$";
    for(unsigned int i=0;i<e.size();++i)
        cout<<e[i].first<<","<<e[i].second<<":";
    cout<<"$"<<endl;
}

void print(const vector<string>& e, ofstream& save)
{
    save<<"$"<<endl;
    for(unsigned int i=0;i<e.size();++i)
    {
        save<<e[i]<<"|";
        if(i%50==49)
            save<<endl;
    }
    save<<"$"<<endl;
}

void print(const vector<pair<string,string> >& e, ofstream& save)
{
    save<<"$"<<endl;
    for(unsigned int i=0;i<e.size();i++)
    {
        save<<e[i].first<<","<<e[i].second<<"|";
        if(i%25==24)
            save<<endl;
    }
    save<<"$"<<endl;
}
```

`fonction.cpp` (fonctions de détermination des caractéristiques des fonctions de m-configuration)

```
bool is_m_function(const string& m_conf){
    bool arg=false;
    int niveau=0;
    long unsigned int n=m_conf.size();

    for(unsigned int i=0;i<n;++i){
        if(m_conf[i]=='('){
            if(i==0)
                return false;
            arg=true;
            ++niveau;
        }
        else if(m_conf[i]==')')
            --niveau;
    }
    return (niveau==0 && arg);
}

unsigned int arite_m_conf(const string& m_conf){
    unsigned int comp=0;
    int niveau=0;
    long unsigned int n=m_conf.size();

    for(unsigned int i=0;i<n;++i){
        if(m_conf[i]=='(')
            ++niveau;
        else if(m_conf[i]==')')
            --niveau;
        else if(m_conf[i]==',' && niveau==1)
            ++comp;
    }
    return comp+1;
}

string name_m_conf(const string& m_conf){
    string sortie;
    long unsigned int n=m_conf.size();

    for(unsigned int i=0;i<n;++i){
        if(m_conf[i]=='(')
            break;
        sortie+=m_conf[i];
    }
    return sortie;
}
```

Cette fonction décompose une chaîne représentant une m-fonction en un tableau de chaînes. La première valeur du tableau est le nom de la m-fonction. Les cases suivantes contiennent les noms des arguments de la m-configuration.

`fonction.cpp` (analyse lexicale d'une m-fonction)

```
90  vector<string> analyse_syntax_m_conf(const string& m_conf){
91      vector<string> sortie(0);
92      string work;
93      unsigned int i=0;
94      int niveau=1;
95      long unsigned int n=m_conf.size();
96
97      work.clear();
98      while(i<n){
99          if(m_conf[i]=='(')
100             break;
101         work+=m_conf[i];
102         ++i;
103     }
104     sortie.push_back(work);
105     ++i;
106
107     while(i<n){
108         work.clear();
109         while((m_conf[i]!=',' && m_conf[i]!=')') || niveau>1){
110             if(m_conf[i]=='(')
111                 ++niveau;
112             else if(m_conf[i]==')')
113                 --niveau;
114             work+=m_conf[i];
115             ++i;
116         }
117         ++i;
118         sortie.push_back(work);
119     }
120     return sortie;
121 }
```

On décompose une liste de symboles sous la forme d'un tableau.
Il y a trois cas :

— symbole=="" ce qui correspond à une absence de filtrage, on renvoie un tableau vide.
— symbole=="("+c+")" signifiant que seuls les symboles de c sont acceptés, on renvoie un tableau contenant le nom des valeurs.
— symbole=="non("+c+")" signifiant que seuls les symboles de c sont acceptés, on renvoie un tableau contenant le nom des valeurs précédées de "non" (dans la première case).

`fonction.cpp`

```
122 vector<string> analyse_syntax_symbole(const string& symbole){
123     if(symbole.size()==0) ///Si tous les symboles sont acceptes
124         return vector<string>(0);
125 
126     vector<string> sortie(0); /* Sinon on fait une analyse syntaxique similaire
127         a celle des m_configurations */
128     string work;
129     unsigned int i=0;
130     int niveau=1;
131     long unsigned int n=symbole.size();
132 
133     while(i<n){
134         if(symbole[i]=='(')
135             break;
136         work=work+symbole[i];
137         ++i;
138     }
139     if(work.size()>0)
140         sortie.push_back(work);
141     ++i;
142 
143     while(i<n){
144         work.clear();
145         while((symbole[i]!=',' && symbole[i]!=')') || niveau>1){
146             if(symbole[i]=='(')
147                 ++niveau;
148             else if(symbole[i]==')')
149                 --niveau;
150             work+=symbole[i];
151             ++i;
152         }
153         ++i;
154         sortie.push_back(work);
155     }
156     return sortie;
```

```
157 }
```

Cette fonction teste si un motif de filtrage sur les symboles accepte un symbole donné. Le filtrage est d'abord décomposé avec la fonction précédente. Si le motif est vide on accepte. Si le motif est un filtrage par exclusion, on vérifie que le symbole n'appartient à aucun de ceux spécifiés. Sinon on vérifie que le symbole testé est bien un de ceux qui sont acceptés.

`fonction.cpp`

```
158 bool analyse_lex_symbole(const string& filtrage, const string& symbole){
159     if(filtrage.size()==0)
160         return true;
161     vector<string> syntaxe=analyse_syntax_symbole(filtrage);
162     long unsigned int n=syntaxe.size();
163 
164     if(n==0)
165         return true;
166 
167     if(syntaxe[0]=="non"){
168         for(unsigned int i=1;i<n;++i)
169             if(syntaxe[i]==symbole)
170                 return false;
171         return true;
172     }
173 
174     for(unsigned int i=0;i<n;++i)
175         if(syntaxe[i]==symbole)
176             return true;
177     return false;
178 
179 }
```

On veut renvoyer la liste des opérations à effectuer sous la forme d'un tableau de couple. Le premier élément contient le type d'opération à effectuer (G,D,E,I) et le deuxième contient l'argument le cas échéant.

On utilise m_conf et m_fonc pour reconnaître quels sont les caractères qui appartiennent au motif de la m-configuration et pour les remplacer par leurs valeurs effectives.

`fonction.cpp`

```
180 vector<pair<string,string> > analyse_syntax_operation(const string& operations,
181         const vector<string>& m_conf, const vector<string>& m_fonc){
182     vector<pair<string,string> > sortie;
183     string work;
184     pair<string,string> w2;
185     long unsigned int n=operations.size();
186 
187     for(unsigned int i=0;i<n;++i){
188         work.clear();
189         while(i<n){
190             if(operations[i]==',')
191                 break;
192             work+=operations[i];
193             ++i;
194         }
195         w2.first=work;
196         sortie.push_back(w2);
197     }
198     long unsigned int m=sortie.size();
199     long unsigned int o=m_fonc.size();
200 
201     for(unsigned int i=0;i<m;++i)
202         if(sortie[i].first[0]=='I')
203             for(unsigned int j=1;j<sortie[i].first.size();++j)
204                 sortie[i].second=sortie[i].second+sortie[i].first[j];
205             sortie[i].first="I";
206 
207     for(unsigned int i=0;i<m;++i)
208         if(sortie[i].first=="I")
209             for(unsigned int j=1;j<o;++j)
210                 if(sortie[i].second==m_fonc[j])
211                 {
212                     sortie[i].second=m_conf[j];
213                     break;
214                 }
215 
216     return sortie;
217 }
```

Cette fonction, à l'écriture un peu plus complexe, permet de donner l'expression développée du nouvel état de la machine.

fonction.cpp

```
string substitution(const vector<string>& m_fonc, const vector<string>& m_conf,
        const vector<string>& result, const string& sym){
    long unsigned int n=result.size();

    if(n==1)
        for(unsigned int i=0;i<m_fonc.size();++i)
            if(result[0]==m_fonc[i])
                return m_conf[i];

    if(n==1)
        return result[0];

    string sortie=result[0]+"(";
    bool found=false;
    for(unsigned int i=1;i<n;++i){
        if(is_m_function(result[i]))
            sortie=sortie+substitution(m_fonc,m_conf,
                    analyse_syntax_m_conf(result[i]),sym);
        else{
            found=false;
            for(unsigned int j=0;j<m_fonc.size();++j){
                if(result[i]=="#"){
                    found=true;
                    sortie=sortie+sym;
                    break;
                }
                else if(result[i]==m_fonc[j]){
                    found=true;
                    sortie=sortie+m_conf[j];
                    break;
                }
            }
            if(not found)
                sortie=sortie+result[i];
        }
        sortie+=",";
    }
    sortie[sortie.size()-1]=')';
    return sortie;
}
```

Cette fonction permet de substituer dans un motif de symboles les symboles effectifs.

`fonction.cpp`

```
257 string substitution_symbole(const vector<string>& m_fonc,
258         const vector<string>& m_conf, const vector<string>& sym){
259     string sortie;
260     bool found=false;
261     bool it_1=false;
262     unsigned int k=0;
263     long unsigned int n=sym.size();
264     long unsigned int m=m_fonc.size();
265 
266     if(n==0)
267         return "";
268 
269     if(sym[0]=="non"){
270         sortie=sortie+"non";
271         k=1;
272     }
273 
274     sortie+='(';
275 
276     for(unsigned int i=k;i<n;++i){
277         found=false;
278         for(unsigned int j=0;j<m;++j)
279             if(sym[i]==m_fonc[j]){
280                 sortie=sortie+m_conf[j];
281                 found=true;
282                 break;
283             }
284         if(not found)
285             sortie+=sym[i];
286         if(not it_1)
287             sortie+=",";
288         it_1=true;
289     }
290     sortie[sortie.size()-1]=')';
291     return sortie;
292 }
```

`transition.h` (Déclaration de la classe abstraite codant les transitions)

```
#ifndef TRANSITION_H_INCLUDED
#define TRANSITION_H_INCLUDED

#include "fonction.h"

class Transition{
public:
    Transition(const std::string m_fonc, const std::string symbole,
        const std::string operation, const std::string m_config_resultante);
    Transition();
    virtual ~Transition();
    virtual void exec(std::vector<std::string>& ruban, unsigned int& pos,
        std::string& m_config, const std::string& B,
        const std::string& m_fonc)=0;
    virtual std::string name()=0;
    virtual unsigned int arite() const=0;
    std::string get_m_config();
    std::string get_symbole();
    virtual bool symbole_match(const std::string& e,
        const std::string& m_config)=0;

protected:
    std::string m_fonc;
    std::string symbole;
    std::string operation;
    std::string m_config_resultante;
};

#endif // TRANSITION_H_INCLUDED
```

`transition.cpp` (fonctions simples)

```cpp
#include "transition.h"

using namespace std;

string Transition::get_m_config(){
    return m_fonc;
}

string Transition::get_symbole(){
    return symbole;
}

Transition::Transition(const string m_fonc_, const string symbole_,
    const string operation_, const string m_config_resultante_) :
m_fonc(m_fonc_), symbole(symbole_), operation(operation_),
    m_config_resultante(m_config_resultante_)
{}

Transition::Transition() :
m_fonc(""), symbole(""), operation(""), m_config_resultante("")
{}

Transition::~Transition()
{}
```

`m_trans.h` (Déclaration de la classe des m-transitions)

```
#ifndef M_TRANS_H_INCLUDED
#define M_TRANS_H_INCLUDED

#include "transition.h"

class MTrans : public Transition{
public:
    MTrans(const std::string m_fonc, const std::string symbole,
            const std::string operation,
            const std::string m_config_resultante);
    virtual void exec(std::vector<std::string>& ruban, unsigned int& pos,
            std::string& m_config, const std::string& B,
            const std::string& m_fonc);
    virtual std::string name();
    virtual unsigned int arite() const __attribute__((pure));
    virtual bool symbole_match(const std::string& e,
            const std::string& m_config);
};

#endif // M_TRANS_H_INCLUDED
```

m_trans.cpp (Fonctions de base)

```cpp
#include "m_trans.h"

using namespace std;

MTrans::MTrans(const string m_fonc_, const string symbolee,
        const string operation_, const string m_config_resultante_) :
m_fonc(m_fonc_), symbole(symbole_), operation(operation_),
        m_config_resultante(m_config_resultante_)
{}

string MTrans::name(){
    return name_m_conf(m_fonc);
}

unsigned int MTrans::arite() const{
    return 0;
}

bool MTrans::symbole_match(const string& e, const string& m_config){
    (void)m_config;
    return analyse_lex_symbole(symbole, e);
}
```

m_trans.cpp (Exécution de la m-transition)

```
23 void MTrans::exec(vector<string>& ruban, unsigned int& pos, string& m_config,
24         const string& B, const string& m_fonction){
25     vector<pair<string,string> > op=analyse_syntax_operation(operation,
26             analyse_syntax_m_conf(m_config),
27             analyse_syntax_m_conf(m_fonction));
28     unsigned int position_prec=pos;
29     long unsigned int n=op.size();
30 
31     for(unsigned int i=0;i<n;++i){
32         if(op[i].first=="E"){
33             ruban[static_cast<size_t>(pos)]=B;
34         }
35         else if(op[i].first=="D"){
36             ++pos;
37             if(ruban.size()<pos+5)
38                 ruban.resize(static_cast<size_t>(pos+30),B);
39         }
40         else if(op[i].first=="G"){
41             pos = (pos > 1 ? pos-1 : 0);
42         }
43         else if(op[i].first=="I"){
44             if(op[i].second=="#"){
45                 string val;
46                 val=ruban[static_cast<size_t>(position_prec)];
47                     ruban[static_cast<size_t>(pos)]=val;
48             }
49             else if(op[i].second==""){
50                 ruban[static_cast<size_t>(pos)]=B;
51             }
52             else{
53                 ruban[static_cast<size_t>(pos)]=op[i].second;
54             }
55         }
56     }
57 
58     string val;
59     val=ruban[static_cast<size_t>(position_prec)];
60 
61     m_config=substitution(analyse_syntax_m_conf(m_fonction),
62                           analyse_syntax_m_conf(m_config),
63                           analyse_syntax_m_conf(m_config_resultante),
64                           val);
65 }
```

`m_fonc.h` (Déclaration de la classe des m-fonctions)

```
#ifndef M_CONF_H_INCLUDED
#define M_CONF_H_INCLUDED

#include "transition.h"

class MFonc : public Transition {
public:
    MFonc(const std::string m_fonc, const std::string symbole,
            const std::string operation,
            const std::string m_config_resultante);
    virtual void exec(std::vector<std::string>& ruban, unsigned int& pos,
            std::string& m_config, const std::string& B,
            const std::string& m_fonc);
    virtual std::string name();
    virtual unsigned int arite() const __attribute__((pure));
    virtual bool symbole_match(const std::string& e,
            const std::string& m_config);
};

#endif // M_CONF_H_INCLUDED
```

m_fonc.cpp (Fonctions de base)

```cpp
#include "m_fonc.h"

using namespace std;

MFonc::MFonc(const string m_fonc_, const string symbole_,
        const string operation_, const string m_config_resultante_) :
m_fonc(m_fonc_), symbole(symbole_), operation(operation_),
        m_config_resultante(m_config_resultante_)
{}

string MFonc::name(){
    return name_m_conf(m_fonc);
}

unsigned int MFonc::arite() const{
    return arite_m_conf(m_fonc);
}

bool MFonc::symbole_match(const string& e, const string& m_config){
    return analyse_lex_symbole(
            substitution_symbole(analyse_syntax_m_conf(m_fonc),
                                 analyse_syntax_m_conf(m_config),
                                 analyse_syntax_symbole(symbole)),
                               e);
}
```

m_fonc.cpp (Exécution de la m-fonction)

```
26 void MFonc::exec(vector<string>& ruban, unsigned int& pos, string& m_config,
27         const string& B, const string& m_fonction){
28     vector<pair<string,string> > op=analyse_syntax_operation(operation,
29             analyse_syntax_m_conf(m_config),
30             analyse_syntax_m_conf(m_fonction));
31     unsigned int position_prec=pos;
32     long unsigned int n=op.size();
33 
34     for(unsigned int i=0;i<n;++i){
35         if(op[i].first=="E"){
36             ruban[static_cast<size_t>(pos)]=B;
37         }
38         else if(op[i].first=="D"){
39             ++pos;
40             if(ruban.size() < pos+5)
41                 ruban.resize(static_cast<size_t>(pos+30),B);
42         }
43         else if(op[i].first=="G"){
44             pos=(pos > 1 ? pos-1 : 0);
45         }
46         else if(op[i].first=="I"){
47             if(op[i].second=="#"){
48                 string val;
49                 val=ruban[static_cast<size_t>(position_prec)];
50                 ruban[static_cast<size_t>(pos)]=val;
51             }
52             else if(op[i].second==""){
53                 ruban[static_cast<size_t>(pos)]=B;
54             }
55             else{
56                 ruban[static_cast<size_t>(pos)]=op[i].second;
57             }
58         }
59     }
60 
61     string val;
62     val=ruban[static_cast<size_t>(position_prec)];
63 
64     m_config=substitution(analyse_syntax_m_conf(m_fonction),
65                           analyse_syntax_m_conf(m_config),
66                           analyse_syntax_m_conf(m_config_resultante),
67                           val);
68 
69 }
```

`machine.h` (déclaration de la classe simulant une machine de TURING)

```
#ifndef MACHINE_H_INCLUDED
#define MACHINE_H_INCLUDED

#include <ctime>
#include "transition.h"

class Transition;
class MFonc;
class MTrans;

class TM
{
public:
    TM();
    TM(std::string q0, std::string B);
    ~TM();
    void exec(std::vector<std::string> ruban);
    void exec(std::vector<std::string> ruban, unsigned int p);
    void exec();
    std::vector<std::string> getRuban() const;
    void setRuban(const std::vector<std::string>& e);
    std::vector<Transition*> getTable() const;
    void setTable(const std::vector<Transition*>& e);
    void addTable(Transition* e);
    void addTable(const std::vector<Transition*>& e);
    std::string getB() const;
    void setB(const std::string& e);
    std::string getQ0() const;
    void setQ0(const std::string& e);
    unsigned int getPos() const __attribute__((pure));
    void setPos(unsigned int e);

protected:
    std::string B;
    std::string q0;
    std::vector<Transition*> table;
    std::vector<std::string> ruban;
    unsigned int pos;
    std::string m_configuration;
};

#endif // MACHINE_H_INCLUDED
```

machine.cpp (fonctions simples)

```cpp
#include "machine.h"
#include "fonction.h"
#include <iostream>
#include <fstream>
#define EVER ;;
#define IT_MAX 1000000000 //Record atteint en plus de 6h de simulation

using namespace std;

TM::TM() :
B(""), q0(""), table(vector<Transition*>()), ruban(vector<string>()), pos(0),
        m_configuration("")
{}

TM::TM(string q0_, string B_) :
B(B_), q0(q0_), table(vector<Transition*>()), ruban(vector<string>()), pos(0),
        m_configuration("")
{}

TM::~TM(){
    for(unsigned int i=0;i<table.size();++i)
        delete table[i];
}

vector<string> TM::getRuban() const{
    return ruban;
}

void TM::setRuban(const vector<string>& e){
    ruban=e;
}

vector<Transition*> TM::getTable() const{
    return table;
}

void TM::setTable(const vector<Transition*>& e){
    table=e;
}

void TM::addTable(Transition* e){
    table.push_back(e);
}
```

machine.cpp (fonctions simples)

```
44 void TM::addTable(const vector<Transition*>& e){
45     for(unsigned int i=0;i<e.size();++i)
46         table.push_back(e[i]);
47 }
48
49 string TM::getB() const{
50     return B;
51 }
52
53 void TM::setB(const string& e){
54     B=e;
55 }
56
57 string TM::getQ0() const{
58     return q0;
59 }
60
61 void TM::setQ0(const string& e){
62     q0=e;
63 }
64
65 unsigned int TM::getPos() const{
66     return pos;
67 }
68
69 void TM::setPos(unsigned int e){
70     pos=e;
71 }
72
73 void TM::exec(vector<string> ruban_, unsigned int p){
74     ruban=ruban_;
75     pos=p;
76     exec();
77 }
```

machine.cpp (fonction principale simulant l'exécution)

```
78  void TM::exec(){
79      int it=0;
80      time_t t1, t2;
81      double dif;
82      long unsigned int n=table.size();
83      time(&t1);
84      if(ruban.size()<pos)
85          ruban.resize(pos+10);
86      bool found=false;
87      m_configuration=q0;
88      unsigned int t;
89      ofstream save("save_file.txt");
90      for(EVER){
91          found=false;
92          if(is_m_function(m_configuration))
93              for(unsigned int i=0;i<n;++i)
94                  if(table[i]->name()==name_m_conf(m_configuration) &&
95                  table[i]->arite()==arite_m_conf(m_configuration) &&
96                  table[i]->symbole_match(ruban[pos],m_configuration)){
97                      t=i;found=true;break;}
98          else
99              for(unsigned int i=0;i<n;++i)
100                 if(table[i]->name()==m_configuration &&
101                 table[i]->symbole_match(ruban[pos],m_configuration)){
102                     t=i;found=true;break;}
103         if(not found) break;
104         save<<ruban.size()<<endl;
105         table[t]->exec(ruban,pos,m_configuration,B,table[t]->get_m_config());
106         save<<ruban.size()<<endl;
107         print(ruban,save);
108         save<<pos<<" "<<ruban[pos]<<endl<<it<<" "<<m_configuration<<endl<<endl;
109         if(it>IT_MAX) break;
110         if(it%1000==0) cout<<it/1000<<" ";
111         ++it;
112     }
113     time(&t2);
114     dif = difftime (t2,t1);
115     save<<"Time : "<<dif<<endl<<"it : "<<it<<endl;
116     for(unsigned int i=0;i<ruban.size();i++){
117         save<<ruban[i]<<"|";
118         if(i%50==0)
119             save<<endl;
120     }
121     save<<endl<<pos<<" "<<ruban[pos]<<endl<<m_configuration<<endl;
122     save.close();
123 }
```

UTM.h (générateur de machine de TURING universelle)

```
#ifndef UTM_H
#define UTM_H

#include "fonction.h"
#include "machine.h"
#include "m_trans.h"
#include "m_fonc.h"

class UTM{
public:
    UTM(){};
    TM* newUTM();
    std::vector<std::string> newUTMTape();

private:
    std::vector<Transition*> nouvelleTableFind();
    std::vector<Transition*> nouvelleTableErase();
    std::vector<Transition*> nouvelleTablePrintAtEnd();
    std::vector<Transition*> nouvelleTableFindAndCopy();
    std::vector<Transition*> nouvelleTableCopyErase();
    std::vector<Transition*> nouvelleTableReplace();
    std::vector<Transition*> nouvelleTableCR();
    std::vector<Transition*> nouvelleTableCompareErase();
    std::vector<Transition*> nouvelleTableCompare();
    std::vector<Transition*> nouvelleTableQ();
    std::vector<Transition*> nouvelleTableCopyEraseMultiple();
    std::vector<Transition*> nouvelleTableCopyE();
    std::vector<Transition*> nouvelleTableCon();
    std::vector<Transition*> nouvelleTableUtmInit();
    std::vector<Transition*> nouvelleTableUtmAnf();
    std::vector<Transition*> nouvelleTableUtmSim();
    std::vector<Transition*> nouvelleTableUtmMf();
    std::vector<Transition*> nouvelleTableUtmSh();
    std::vector<Transition*> nouvelleTableUtmInst();
};

#endif // UTM_H
```

UTM.cpp (création de tables)

```cpp
vector<Transition*> UTM::nouvelleTableFind(){ // Find
    vector<Transition*> sortie;

    sortie.push_back(new MFonc("f(EE,BB,aa)","(@)","G","f1(EE,BB,aa)"));
    sortie.push_back(new MFonc("f(EE,BB,aa)","non(@,o)","G","f(EE,BB,aa)"));
    sortie.push_back(new MFonc("f(EE,BB,aa)","(o)","G","f(EE,BB,aa)"));
    sortie.push_back(new MFonc("f1(EE,BB,aa)","(aa)","N","EE"));
    sortie.push_back(new MFonc("f1(EE,BB,aa)","(o)","D","f2(EE,BB,aa)"));
    sortie.push_back(new MFonc("f1(EE,BB,aa)","non(o,aa)","D","f1(EE,BB,aa)"));
    sortie.push_back(new MFonc("f2(EE,BB,aa)","(aa)","N","EE"));
    sortie.push_back(new MFonc("f2(EE,BB,aa)","(o)","D","BB"));
    sortie.push_back(new MFonc("f2(EE,BB,aa)","non(o,aa)","D","f1(EE,BB,aa)"));

    return sortie;
}

vector<Transition*> UTM::nouvelleTableErase(){ // Erase
    vector<Transition*> sortie;

    sortie.push_back(new MFonc("e(EE,BB,aa)","","","f(e1(EE,BB,aa),BB,aa)"));
    sortie.push_back(new MFonc("e1(EE,BB,aa)","","E","EE"));
    sortie.push_back(new MFonc("e(BB,aa)","","","e(e(BB,aa),BB,aa)"));

    return sortie;
}

vector<Transition*> UTM::nouvelleTablePrintAtEnd(){ // Print at end
    vector<Transition*> sortie;

    sortie.push_back(new MFonc("pe(EE,bb)","","","f(pe1(EE,bb),EE,@)"));
    sortie.push_back(new MFonc("pe1(EE,bb)","non(o)","D,D","pe1(EE,bb)"));
    sortie.push_back(new MFonc("pe1(EE,bb)","(o)","Ibb","EE"));

    return sortie;
}

vector<Transition*> UTM::nouvelleTableCopyErase(){ // Copy erase
    vector<Transition*> sortie;

    sortie.push_back(new MFonc("ce(EE,BB,aa)","","","c(e(EE,BB,aa),BB,aa)"));
    sortie.push_back(new MFonc("ce(BB,aa)","","","ce(ce(BB,aa),BB,aa)"));

    return sortie;
}
```

UTM.cpp (création de tables)

```
45  vector<Transition*> UTM::nouvelleTableFindAndCopy(){ // Print and copy
46      vector<Transition*> sortie;
47      sortie.push_back(new MFonc("l(EE)","","G","EE"));
48      sortie.push_back(new MFonc("r(EE)","","D","EE"));
49      sortie.push_back(new MFonc("f'(EE,BB,aa)","","","f(l(EE),BB,aa)"));
50      sortie.push_back(new MFonc("f''(EE,BB,aa)","","","f(r(EE),BB,aa)"));
51      sortie.push_back(new MFonc("c(EE,BB,aa)","","","f'(c1(EE),BB,aa)"));
52      sortie.push_back(new MFonc("c1(EE)","","","pe(EE,#)"));
53
54      return sortie;
55  }
56
57  vector<Transition*> UTM::nouvelleTableReplace(){ // Replace
58      vector<Transition*> sortie;
59      sortie.push_back(new MFonc("re(EE,BB,aa,bb)","","",
60              "f(re1(EE,BB,aa,bb),BB,aa)"));
61      sortie.push_back(new MFonc("re1(EE,BB,aa,bb)","","E,Ibb","EE"));
62      sortie.push_back(new MFonc("re(BB,aa,bb)","","E,Ibb",
63              "re(re(BB,aa,bb),BB,aa,bb)"));
64
65      return sortie;
66  }
67
68  vector<Transition*> UTM::nouvelleTableCR(){
69      vector<Transition*> sortie;
70      sortie.push_back(new MFonc("cr(EE,BB,aa)","","",
71              "c(re(EE,BB,aa,$a$),BB,aa)"));
72      sortie.push_back(new MFonc("cr(BB,aa)","","",
73              "cr(ce(BB,aa),re(BB,$a$,aa),aa)"));
74
75      return sortie;
76  }
77
78  vector<Transition*> UTM::nouvelleTableCompare(){ // Compare
79      vector<Transition*> sortie;
80      sortie.push_back(new MFonc("cp(EE1,UU,EE,aa,bb)","","",
81              "f'(cp1(EE1,UU,bb),f(UU,EE,bb),aa)"));
82      sortie.push_back(new MFonc("cp1(EE,UU,bb)","","",
83              "f'(cp2(EE,UU,#),UU,bb)"));
84      sortie.push_back(new MFonc("cp2(EE,UU,gg)","(gg)","","EE"));
85      sortie.push_back(new MFonc("cp2(EE,UU,gg)","non(gg,o)","","UU"));
86
87      return sortie;
88  }
```

UTM.cpp (création de tables)

```
89  vector<Transition*> UTM::nouvelleTableCompareErase(){ // Compare erase
90      vector<Transition*> sortie;
91
92      sortie.push_back(new MFonc("cpe(EE1,UU,EE,aa,bb)","","",
93              "cp(e(e(EE1,EE,bb),EE,aa),UU,EE,aa,bb)"));
94      sortie.push_back(new MFonc("cpe(UU,EE,aa,bb)","","",
95              "cpe(cpe(UU,EE,aa,bb),UU,EE,aa,bb)"));
96
97      return sortie;
98  }
99
100 vector<Transition*> UTM::nouvelleTableQ(){
101     vector<Transition*> sortie;
102
103     sortie.push_back(new MFonc("q(EE)","non(o)","D","q(EE)"));
104     sortie.push_back(new MFonc("q(EE)","(o)","D","q1(EE)"));
105     sortie.push_back(new MFonc("q1(EE)","non(o)","D","q(EE)"));
106     sortie.push_back(new MFonc("q1(EE)","(o)","","EE"));
107     sortie.push_back(new MFonc("q(EE,aa)","","","q(q1(EE,aa))"));
108     sortie.push_back(new MFonc("q1(EE,aa)","(aa)","","EE"));
109     sortie.push_back(new MFonc("q1(EE,aa)","non(aa)","G","q1(EE,aa)"));
110
111     return sortie;
112 }
113
114 vector<Transition*> UTM::nouvelleTableCopyEraseMultiple(){
115     vector<Transition*> sortie;
116
117     sortie.push_back(new MFonc("pe2(EE,aa,bb)","","","pe(pe(EE,bb),aa)"));
118     sortie.push_back(new MFonc("ce2(BB,aa,bb)","","","ce(ce(BB,bb),aa)"));
119     sortie.push_back(new MFonc("ce3(BB,aa,bb,gg)","","",
120             "ce(ce2(BB,bb,gg),aa)"));
121     sortie.push_back(new MFonc("ce4(BB,aa,bb,gg,dd)","","",
122             "ce(ce3(BB,bb,gg,dd),aa)"));
123     sortie.push_back(new MFonc("ce5(BB,aa,bb,gg,dd,ee)","","",
124             "ce(ce4(BB,bb,gg,dd,ee),aa)"));
125
126     return sortie;
127 }
```

UTM.cpp (création de tables)

```
128 vector<Transition*> UTM::nouvelleTableCopyE(){
129     vector<Transition*> sortie;
130 
131     sortie.push_back(new MFonc("e(EE)","(@)","D","e1(EE)"));
132     sortie.push_back(new MFonc("e(EE)","non(@)","G","e(EE)"));
133     sortie.push_back(new MFonc("e1(EE)","non(o)","D,E,D","e1(EE)"));
134     sortie.push_back(new MFonc("e1(EE)","(o)","","EE"));
135 
136     return sortie;
137 }
138 
139 vector<Transition*> UTM::nouvelleTableCon(){ // Configuration
140     vector<Transition*> sortie;
141 
142     sortie.push_back(new MFonc("con(EE,aa)","non(A,o)","D,D","con(EE,aa)"));
143     sortie.push_back(new MFonc("con(EE,aa)","(A)","G,Iaa,D","con1(EE,aa)"));
144     sortie.push_back(new MFonc("con1(EE,aa)","(A)","D,Iaa,D","con1(EE,aa)"));
145     sortie.push_back(new MFonc("con1(EE,aa)","(C)","D,Iaa,D","con2(EE,aa)"));
146     sortie.push_back(new MFonc("con1(EE,aa)","(o)","IC,D,Iaa,D,D,D","EE"));
147     sortie.push_back(new MFonc("con2(EE,aa)","(B)","D,Iaa,D","con2(EE,aa)"));
148     sortie.push_back(new MFonc("con2(EE,aa)","non(B)","D,D","EE"));
149 
150     return sortie;
151 }
152 
153 vector<Transition*> UTM::nouvelleTableUtmInit(){
154     vector<Transition*> sortie;
155 
156     sortie.push_back(new MTrans("b","","","f(b1,b1,+)"));///Begin
157     sortie.push_back(new MTrans("b1","","D,D,I:,D,D,IC,D,D,IA","anf"));
158             ///Inscrire :CA
159     return sortie;
160 }
161 
162 vector<Transition*> UTM::nouvelleTableUtmAnf(){
163     vector<Transition*> sortie;
164 
165     sortie.push_back(new MTrans("anf","","","q(anf1,:)"));
166     sortie.push_back(new MTrans("anf1","","","con(fom,y)"));
167     sortie.push_back(new MTrans("fom","(;)","D,Iz,G","con(fmp,x)"));
168     sortie.push_back(new MTrans("fom","(z)","G,G","fom"));
169     sortie.push_back(new MTrans("fom","non(;,z)","G","fom"));
170     sortie.push_back(new MTrans("fmp","","","cpe(e(e(anf,y),x),sim,x,y)"));
171 
172     return sortie;
173 }
```

UTM.cpp (création de tables)

```
174 vector<Transition*> UTM::nouvelleTableUtmSim(){ // Simulate
175     vector<Transition*> sortie;
176 
177     sortie.push_back(new MTrans("sim","","","f'(sim1,sim1,z)"));
178     sortie.push_back(new MTrans("sim1","","","con(sim2,o)"));
179     sortie.push_back(new MTrans("sim2","(A)","","sim3"));
180     sortie.push_back(new MTrans("sim2","non(A)","G,Iu,D,D,D","sim2"));
181     sortie.push_back(new MTrans("sim3","non(A)","G,Iy","e(mf,z)"));
182     sortie.push_back(new MTrans("sim3","(A)","G,Iy,D,D,D","sim3"));
183 
184     return sortie;
185 }
186 
187 vector<Transition*> UTM::nouvelleTableUtmMf(){
188     vector<Transition*> sortie;
189 
190     sortie.push_back(new MTrans("mf","","","q(mf1,:)"));
191     sortie.push_back(new MTrans("mf1","non(A)","D,D","mf1"));
192     sortie.push_back(new MTrans("mf1","(A)","G,G,G,G","mf2"));
193     sortie.push_back(new MTrans("mf2","(B)","D,Ix,G,G,G","mf2"));
194     sortie.push_back(new MTrans("mf2","(:)","","mf4"));
195     sortie.push_back(new MTrans("mf2","(C)","D,Ix,G,G,G","mf3"));
196     sortie.push_back(new MTrans("mf3","non(:)","D,Iv,G,G,G","mf3"));
197     sortie.push_back(new MTrans("mf3","(:)","","mf4"));
198     sortie.push_back(new MTrans("mf4","","","con(l(l(mf5)),)"));
199     sortie.push_back(new MTrans("mf5","non(o)","D,Iw,D","mf5"));
200     sortie.push_back(new MTrans("mf5","(o)","I:","sh"));
201 
202     return sortie;
203 }
204 
205 vector<Transition*> UTM::nouvelleTableUtmSh(){
206     vector<Transition*> sortie;
207 
208     sortie.push_back(new MTrans("sh","","","f(sh1,inst,u)"));///Show
209     sortie.push_back(new MTrans("sh1","","G,G,G","sh2"));
210     sortie.push_back(new MTrans("sh2","(C)","D,D,D,D","sh3"));
211     sortie.push_back(new MTrans("sh2","non(C)","","inst"));
212     sortie.push_back(new MTrans("sh3","(B)","D,D","sh4"));
213     sortie.push_back(new MTrans("sh3","non(B)","","inst"));
214     sortie.push_back(new MTrans("sh4","(B)","D,D","sh5"));
215     sortie.push_back(new MTrans("sh4","non(B)","","pe2(inst,0,:)"));
216     sortie.push_back(new MTrans("sh5","(B)","","inst"));
217     sortie.push_back(new MTrans("sh5","non(B)","","pe2(inst,1,:)"));
218 
219     return sortie;
220 }
```

UTM.cpp (création de tables et génération de la machine)

```
221 vector<Transition*> UTM::nouvelleTableUtmInst(){
222     vector<Transition*> sortie;
223 
224     sortie.push_back(new MTrans("inst","","","q(l(inst1),u)"));
225     sortie.push_back(new MTrans("inst1","(G)","D,E","inst1G"));
226     sortie.push_back(new MTrans("inst1","(D)","D,E","inst1D"));
227     sortie.push_back(new MTrans("inst1","(N)","D,E","inst1N"));
228     sortie.push_back(new MTrans("inst1G","","","ce5(ov,v,y,x,u,w)"));
229     sortie.push_back(new MTrans("inst1D","","","ce5(ov,v,x,u,y,w)"));
230     sortie.push_back(new MTrans("inst1N","","","ce5(ov,v,x,y,u,w)"));
231     sortie.push_back(new MTrans("ov","","","e(anf)"));
232 
233     return sortie;
234 }
235 
236 TM* UTM::newUTM(){ // executer avec machine->exec(ruban,2);
237     TM* machine=new TM("b","o");
238 
239     machine->addTable(nouvelleTableFind());
240     machine->addTable(nouvelleTableErase());
241     machine->addTable(nouvelleTablePrintAtEnd());
242     machine->addTable(nouvelleTableFindAndCopy());
243     machine->addTable(nouvelleTableCopyErase());
244     machine->addTable(nouvelleTableReplace());
245     machine->addTable(nouvelleTableCR());
246     machine->addTable(nouvelleTableCompareErase());
247     machine->addTable(nouvelleTableCompare());
248     machine->addTable(nouvelleTableQ());
249     machine->addTable(nouvelleTableCopyEraseMultiple());
250     machine->addTable(nouvelleTableCopyE());
251     machine->addTable(nouvelleTableCon());
252     machine->addTable(nouvelleTableUtmInit());
253     machine->addTable(nouvelleTableUtmAnf());
254     machine->addTable(nouvelleTableUtmSim());
255     machine->addTable(nouvelleTableUtmMf());
256     machine->addTable(nouvelleTableUtmSh());
257     machine->addTable(nouvelleTableUtmInst());
258 
259     return machine;
260 }
```

UTM.cpp (création du ruban)

```
261 vector<string> UTM::newUTMTape(){
262     vector<string> ruban(200,"o");
263     string tape="@@;CACCBDCAA;CAACCDCAAA;CAAACCBBDCAAAA;CAAAACCDCA+";
264 
265     for(unsigned int i=0;i<tape.size();++i)
266         ruban[i]=tape[i];
267 
268     return ruban;
269 }
```

main.cpp (lancement de la machine)

```
1  #include "UTM.h"
2  
3  using namespace std;
4  
5  int main()
6  {
7      UTM* genUTM=new UTM();
8      TM* machine=genUTM->newUTM();
9  
10     vector<string> rubanD(genUTM->newUTMTape());
11 
12     machine->exec(rubanD, 2);
13     delete machine;
14 
15     return 0;
16 }
```

Pas moins de 1 000 000 000 transitions de $\mathcal{U}$ sont nécessaires pour simuler 60 transitions de $\mathcal{M}$ et ce pour une machine $\mathcal{M}$ très simple (4 transitions, 4 états et 2 symboles) et en 6 h 29 min 57 s sur un processeur Intel Core i7-3610QM cadencé à 2.30GHz...

Le programme de simulation des machines de TURING fait plus de 1 000 lignes en C++. Ce programme est capable de simuler des machines écrites avec des tables abrégées avec des m-fonctions à raison de plus de 42 000 transitions par seconde. Un code efficace pour la simulation fait appel à des notions avancées de C++ et demanderait un temps bien supérieur sans cela. D'où l'impossibilité pour TURING de tester sa machine et par conséquent de déceler ses erreurs (cependant les correcteurs successifs, eux, sont inexcusables).

On peut constater que l'existence d'une machine universelle n'a qu'un intérêt théorique. En effet la simulation prend un très grand nombre de transitions pour n'en simuler qu'une. De plus ce nombre est rapidement croissant à cause de l'augmentation de la quantité de données sur le ruban.

Ainsi l'exhibition d'une machine de TURING universelle n'est pas une méthode efficace pour prouver qu'un modèle de calcul est TURING-équivalent. On préfèrera en général exhiber une méthode de construction générale pour exécuter toutes les fonctions.

1.7 Les machines de TURING à plusieurs rubans

On peut aussi définir des machines de TURING à plusieurs rubans. Ces machines présentent un intérêt pour les démonstrations des équivalences des différents modèles. Il est souvent plus simple de montrer l'équivalence d'un modèle avec une machine à plusieurs rubans. Ensuite, il est immédiat que ces machines sont équivalentes aux machines de TURING.

Définition 6 (Machine de TURING à plusieurs rubans)**.**

Une machine de TURING à k rubans est un 7-uplet $(Q, \Gamma, B, \Sigma, q_0, \delta, F)$ où :

- Q est un ensemble fini d'états,
- Γ est l'alphabet des symboles de la bande,
- $B \in \Gamma$ est un symbole particulier dit blanc,
- Σ est l'alphabet des symboles en entrée ($\Sigma \subseteq \Gamma \setminus \{B\}$),
- $q_0 \in Q$ est l'état initial,
- $\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \left\{ \{\longleftarrow, \longrightarrow, \bigcirc\}^k, \{\texttt{HALT}\} \right\}$ est la fonction de transition,
- $F \subseteq Q$ est l'ensemble des états acceptants (dits aussi finaux ou terminaux).

Elles fonctionnent de la même façon que les machines de TURING mais la fonction de transition renvoie un symbole et un mouvement par ruban en plus du nouvel état de la machine étant donné l'état de la machine et le symbole présent sous la tête de lecture de chaque ruban. On introduit pour ça l'absence de mouvement noté $\bigcirc$.

Ce modèle donne une nouvelle liberté dans la façon d'utiliser une machine de TURING. Il est courant de voir des rubans d'entrée et sortie. Selon les versions, ces rubans sont soumis à des contraintes plus ou moins fortes. On impose en général qu'on écrit pas sur le ruban d'entrée et qu'on ne lit pas le ruban de sortie. On peut aussi imposer que le mouvement de la tête de lecture sur ces rubans consiste en un déplacement dans un sens à chaque nouvelle lecture ou écriture, sans jamais revenir en arrière.

Chapitre 2

Les automates cellulaires

On voit ici un modèle de calcul d'un autre genre. Les machines de TURING sont un modèle automatique. On peut le voir sous la forme d'une machine qui réalise des opérations. La proximité avec un ordinateur est immédiate. Les automates cellulaires qu'on voit ici sont un peu différents. Leur mémoire a une structure similaire à celle d'une machine de Turing (bien qu'on puisse ajouter des dimensions) et l'exécution fonctionne par pas de temps, mais la mise à jour de l'état se fait sur toute la mémoire en même temps. Ce modèle est donc davantage parallèle est moins automatique. Nous verrons par la suite de nouveau des modèles proches des machines de TURING mais aussi des modèles de moins en moins automatique jusqu'à atteindre des modèles mathématiques où il n'y a plus de notion de pas de calcul ou de mémoire.

2.1 Définition des automates cellulaires

Définition 7 (Réseau)**.**

Un réseau Λ de $\mathbb{R}^n$ est un sous-groupe discret de $\mathbb{R}^n$ pour l'addition, tel que $\mathrm{Vect}_{\mathbb{R}}(\Lambda) = \mathbb{R}^n$.

Proposition 4.

Pour tout $n \in \mathbb{N}^*$, $\mathbb{Z}^n$ est un réseau de $\mathbb{R}^n$.

Démonstration. Soit $n \in \mathbb{N}^*$.
On a trivialement que $\mathbb{Z}^n$ est un sous-groupe additif de $\mathbb{R}^n$.

De plus, on vérifie que la base canonique de $\mathbb{R}^n$ appartient à $\mathbb{Z}^n$, donc

$$\mathbb{R}^n \subseteq \text{Vect}_{\mathbb{R}}\left(\mathbb{Z}^n\right)$$

Et comme $\mathbb{Z}^n \subset \mathbb{R}^n$, on a

$$\text{Vect}_{\mathbb{R}}\left(\mathbb{Z}^n\right) \subseteq \text{Vect}_{\mathbb{R}}\left(\mathbb{R}^n\right) = \mathbb{R}^n$$
$$\text{d'où } \text{Vect}_{\mathbb{R}}\left(\mathbb{Z}^n\right) = \mathbb{R}^n$$

$\mathbb{Z}^n$ est donc bien un réseau de $\mathbb{R}^n$

□

Nous utiliserons par la suite principalement le réseau $\mathbb{Z}^2$.

Nous allons tout d'abord donner une définition courante d'un automate cellulaire.

Définition 8 (Automate cellulaire)**.**

On appelle automate cellulaire tout 4-uplet (d, Q, V, δ) où :
- $d \in \mathbb{N}^*$, est la dimension de l'automate cellulaire, son réseau est alors $\mathbb{Z}^d$,
- Q est un ensemble fini appelé alphabet, les éléments de Q sont appelés états de l'automate cellulaire,
- $V \subsetneq \mathbb{Z}^d$ fini, est le voisinage. On pose $V = \{v_1, \ldots, v_a\}$,
- $\delta : Q^a \to Q$ est sa règle locale, où $a = |V|$ (arité de l'automate).

On appelle configuration une fonction de $\mathbb{Z}^d \to Q$.
On définit aussi une fonction globale d'évolution de l'automate F_δ :

$$\begin{aligned} F_\delta : Q^{\mathbb{Z}^d} &\to Q^{\mathbb{Z}^d} \\ c &\mapsto \left(x \mapsto \delta\left(c\left(x + v_1\right), \ldots, c\left(x + v_a\right)\right)\right) \end{aligned}$$

Cette fonction permet d'obtenir le nouvel état de l'automate selon la règle locale.

On dispose ainsi d'une définition formelle d'un automate cellulaire bien que celle-ci ne soit pas intuitive ni optimale dans son utilisation pour l'étude des propriétés spécifiques au jeu de la vie de CONWAY, automate qu'on étudiera en particulier. On sera donc amené à procéder à une nouvelle définition spécifique au jeu de la vie.

Définition 9.

Pour $(n, d) \in (\mathbb{N}^*)^2$ on définit la norme n, notée $\|\_\|_n$, par :

$$\forall (X, Y) \in \mathbb{Z}^d \times \mathbb{Z}^d, \left(\sum_{i=1}^{d} |x_i - y_i|^n \right)^{\frac{1}{n}} = \|X - Y\|_n$$

où $X = (x_1, \ldots, x_d)$ et $Y = (y_1, \ldots, y_d)$.
On définit également la norme infinie, $\|\_\|_\infty$, par :

$$\forall (X, Y) \in \mathbb{Z}^d \times \mathbb{Z}^d, \max_{i \in [\![1,d]\!]} (|x_i - y_i|) = \|X - Y\|_\infty$$

Définition 10 (Voisinage de VON NEUMANN et de MOORE)**.**

Pour un automate cellulaire (d, Q, V, δ) on définit le voisinage de VON NEUMANN par :

$$\left\{ z \in \mathbb{Z}^d \;\middle|\; \|z\|_1 \leqslant 1 \right\}$$

et le voisinage de MOORE par :

$$\left\{ z \in \mathbb{Z}^d \;\middle|\; \|z\|_\infty \leqslant 1 \right\}$$

Proposition 5.

Les voisinages de VON NEUMANN et de MOORE sont des voisinages au sens des automates cellulaires.

Dans la suite, on appelle une configuration finie, une configuration qui comporte un nombre cofinie de cellules dans un état particulier considéré comme nul, neutre, mort...Cet état dépend de l'automate étudié mais il sera habituellement noté 0 ou *mort*.

Un automate cellulaire, utilisé en temps que modèle de calcul, est difficile à interprété dans le cas général. Tout d'abord, l'automate ne termine pas. Mais on peut souvent faire un automate qui est stable ou périodique une fois le calcul fini. Il reste le problème du stockage de l'information. Pour cela, il y a deux

façon de faire. La première consiste à utiliser des structures stables ou périodiques qui marque de l'information selon un motif donné. L'autre façon de faire consiste directement à exploité l'état des cellules. Alors, il est nécessaire que les cellules en questions soient stables. On peut alors représenter de façon « graphique » la mémoire d'un autre modèle de calcul, comme une pile, un ruban....

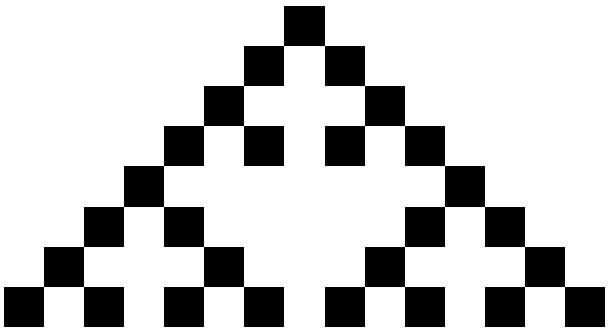
Pour faire un tel usage d'un automate fini, il est nécessaire que la configuration initiale soit finie. On a ainsi une configuration finie à chaque étape. Dans le cas contraire, cela revient à avoir une quantité infinie de données.

2.2 Exemples

Exemple 1.

On peut construire un automate cellulaire dont les état successifs se rapprochent du triangle de SIERPIŃSKI. Pour cet exemple, on prend $d = 1$ (l'automate est dit unidimensionnel), $Q = \{0, 1\}$, $V = \{-1, 0, 1\}$, et

$$\begin{aligned} \delta : Q^3 &\to Q \\ (a, b, c) &\mapsto (a \oplus c) \wedge \neg b \end{aligned}$$



Pour plus de visibilité sur la représentation, l'alphabet a été remplacé par $\{\ , \blacksquare\}$. Cette succession de configurations a été obtenue pour une configuration initiale comprenant un unique 1. On peut en fait constater que la figure construite à partir des configurations successives présente une nature fractale similaire pour toute configuration initiale (bien qu'on puisse convenir que ces fractales n'ont pour la plupart aucune qualité ni esthétique ni mathématique).

Notation 2 (Crochet de IVERSON [Knu92]).

Soit P une proposition. On note $[P]$:

$$[P] = \begin{cases} 1 & \text{si } P \text{ est vraie,} \\ 0 & \text{sinon.} \end{cases}$$

C'est une notation pratique pour décrire des conditions dans une expression. Par exemple, la fonction caractéristique d'un ensemble E est $x \mapsto [x \in E]$.

Proposition 6.

Dans l'automate précédent, en supposant que le carré noir initial est en position 0.

$$\forall n \in \mathbb{N}, \forall p \in \mathbb{N}, \sigma_n(p) = 1 \Rightarrow \left(\forall i \in \mathbb{N}, 2^i \geqslant n \Rightarrow \sigma_{2^i+n}(-2^i + p) = \sigma_{2^i+n}(2^i + p) = 1\right)$$

Exemple 2 (Compteur de majorité)**.**

Ayant ainsi vu un exemple simple d'automate cellulaire, on peut s'intéresser aux plus courants d'entre eux : les bidimensionnels. Ceux-ci sont donc représentables dans l'espace $\mathbb{Z}^2$ et comportent une plus grande variété. Parmi les automates cellulaires connus, on trouve celui défini avec $d = 2$ un alphabet à 2 éléments, son voisinage est celui de MOORE privé de $(0, 0)$ et la cellule prend, dans la configuration suivante, l'état majoritaire dans son voisinage (règle de majorité). Cet automate se stabilise rapidement et ne provoque pas de figure périodique.

Cet automate cellulaire considère le voisinage dans son intégralité sans tenir compte de la position des cellules à l'intérieur du voisinage, on parle d'automate de type *sommatif* puisqu'il ne tient compte que de la population de chaque état représenté dans son voisinage.

De tels automates sont souvent bien simples. Mais nous allons montrer plus loin qu'ils peuvent avoir la même puissance de calcul que les autres automates.

Exemple 3 (Automate de FREDKIN)**.**

On trouve aussi un automate de dimension 2 avec $\{0, 1\}$ comme alphabet et un voisinage de VON NEUMANN privé de $(0, 0)$ utilisant le modèle du compteur de parité. Chaque cellule prend l'état 0 si elle est entourée d'un nombre pair de 1, et prend l'état 1 dans le cas contraire. Il s'agit de l'automate de FREDKIN. Celui-ci possède une propriété particulière : il est capable de dupliquer n'importe quelle configuration initiale finie.

Proposition 7.

Soit $\sigma_0 : (x,y) \in \mathbb{Z}^2 \mapsto [x=0][y=0]$ une configuration initiale de l'automate de FREDKIN.
On note σ_n la configuration de la nième génération de l'automate.
Alors

$$\forall i \in \mathbb{N}, \forall (x,y) \in \mathbb{Z}^2,$$
$$\sigma_{2^i}(x,y) = 1 \Leftrightarrow \left(\exists (\varepsilon_x, \varepsilon_y) \in [\![-1,1]\!]^2 \setminus \{(0,0)\} : (x,y) = \left(\varepsilon_x 2^i, \varepsilon_y 2^i\right)\right)$$

Autrement dit, à la génération 2^i, il n'y a que 8 cellules contenant 1 et elles se trouvent aux coordonnées $(2^i,0)$, $(2^i,2^i)$, $(0,2^i)$, $(-2^i,2^i)$, $(-2^i,0)$, $(-2^i,-2^i)$, $(0,-2^i)$ et $(2^i,-2^i)$.

Démonstration. On définit une opération notée $\oplus$ par

$$\oplus : \left(\{0,1\}^{(\mathbb{Z}^2)}\right)^2 \to \{0,1\}^{(\mathbb{Z}^2)}$$
$$(\sigma,\tau) \mapsto \left((x,y) \in \mathbb{Z}^2 \mapsto (\sigma(x,y) + \tau(x,y))[2]\right)$$

On utilise cette fonction en notation infixe. Cette opération représente la somme point par point du réseau $\mathbb{Z}^2$ à valeur dans $\{0,1\}$, aussi les opérations sont réalisées modulo 2. Il est clair que cette règle est linéaire. En effet, il ne s'agit que d'additions et d'un modulo qui est compatible avec l'addition.

On note F_δ la règle globale de l'automate.

Lemme 1.

Pour toutes configurations σ et τ de l'automate de FREDKIN

$$F_\delta(\sigma \oplus \tau) = F_\delta(\sigma) \oplus F_\delta(\tau)$$

Démonstration. Soit $(x,y) \in \mathbb{Z}^2$. On se donne σ et τ deux configurations initiales de l'automate de FREDKIN et on note v le voisinage de l'automate.

On a

$$\begin{aligned} F_\delta(\sigma \oplus \tau)(x,y) &= \sum_{c \in v(x,y)} (\sigma(c) + \tau(c)) \,[2] \\ &= \left(\sum_{c \in v(x,y)} \sigma(c) + \sum_{c \in v(x,y)} \tau(c) \right) [2] \\ &= \left(\sum_{c \in v(x,y)} \sigma(c)[2] + \sum_{c \in v(x,y)} \tau(c)[2] \right) [2] \\ &= \left(F_\delta(\sigma) \oplus F_\delta(\tau)\right)(x,y) \end{aligned}$$

□

On a donc une propriété de linéarité de la règle globale par rapport à l'opération $\oplus$.

On peut ainsi décomposer toute configuration finie comme une somme (au sens de $\oplus$) finie de configuration nulle sauf en un seul point.

On procède par récurrence sur i :

Initialisation : Les cas $i = 0$ et $i = 1$ sont trivialement vrais.

Hérédité : Soit $n \in \mathbb{N}$, $n \geqslant 1$. On suppose la propriété vrai au rang n.

On a donc 8 cellules qui contiennent un 1 à la $2^{n\text{ème}}$ génération : $(2^n, 0)$, $(2^n, 2^n)$, $(0, 2^n)$, $(-2^n, 2^n)$, $(-2^n, 0)$, $(-2^n, -2^n)$, $(0, -2^n)$ et $(2^n, -2^n)$.

Grâce à la linéarité des configurations de l'automate, on peut sommer le résultat de l'évolution individuelle de ces 8 cellules pendant 2^n générations. La somme permet d'énumérer les cellules contenant 1. On trouve :

Coordonnées	Occurrences
$(0,0)$	8
$(2^n,0)$	4
$(0,2^n)$	4
$(-2^n,0)$	4
$(0,-2^n)$	4
$(2^n,2^n)$	2
$(2^n,-2^n)$	2
$(-2^n,2^n)$	2
$(-2^n,-2^n)$	2
$(0,2^{n+1})$	3
$(2^{n+1},0)$	3
$(0,-2^{n+1})$	3

– Suite sur la page suivante –

Coordonnées	Occurrences
$(-2^{n+1}, 0)$	3
$(2^{n+1}, 2^{n+1})$	1
$(2^{n+1}, -2^{n+1})$	1
$(-2^{n+1}, 2^{n+1})$	1
$(-2^{n+1}, -2^{n+1})$	1
$(2^{n+1}, 2^{n})$	2
$(2^{n}, 2^{n+1})$	2
$(2^{n+1}, -2^{n})$	2
$(2^{n}, -2^{n+1})$	2
$(-2^{n+1}, 2^{n})$	2
$(-2^{n}, 2^{n+1})$	2
$(-2^{n+1}, -2^{n})$	2
$(-2^{n}, -2^{n+1})$	2

Toutes les cellules ayant un nombre d'occurrences impaires seront vivante après la somme modulo 2. À l'inverse, les autres seront morte. On observe bien qu'on obtient la même configuration que précédemment mais d'une taille double.

On a donc bien la propriété au rang $n+1$.

Ainsi, par récurrence, la propriété est vraie à tous les rangs. □

Définition 11.

Pour $(u, v) \in \mathbb{Z}^2$, on définie la translation d'une configuration :

$$\begin{aligned} \tau_{(u,v)} : \{0,1\}^{(\mathbb{Z}^2)} &\to \{0,1\}^{(\mathbb{Z}^2)} \\ \sigma &\mapsto \Big((x,y) \in \mathbb{Z}^2 \mapsto \sigma(x-u, y-v) \Big) \end{aligned}$$

Remarque 1.

Ainsi on peut généraliser le résultat précédent pour toute cellule par translation. En effet un automate cellulaire n'est pas influencé par les coordonnées. Tout résultat vrai à l'origine sera également vrai en tout point.

Proposition 8.

Soit $\sigma_0 : \mathbb{Z}^2 \to \{0,1\}$ une configuration initiale finie de l'automate de FREDKIN.
Soit $M \in \mathbb{N}$ tel que

$$\forall (x,y) \in \mathbb{Z}^2, \|(x,y)\|_\infty \geqslant M \Rightarrow \sigma_0(x,y) = 0$$

On note $k = \min\left\{i \in \mathbb{N} \,\middle|\, 2^i \geqslant M\right\} = \lceil \log_2(M) \rceil$ et on note σ_n la configuration de la nième génération de l'automate.
Alors

$$\forall i \in [\![k, +\infty]\!], \forall (x,y) \in \mathbb{Z}^2, \forall (\varepsilon_x, \varepsilon_y) \in [\![-1,1]\!]^2 \setminus \{(0,0)\},$$
$$\|(x,y)\|_\infty \leqslant M \Rightarrow \sigma_0(x,y) = \sigma_{2^i}\left(\varepsilon_x 2^i + x, \varepsilon_y 2^i + y\right)$$

Démonstration. On note

$$\begin{aligned} \sigma_{(i,j)} : \mathbb{Z}^2 &\to \{0,1\} \\ (x,y) &\mapsto [x = i][y = j] \end{aligned}$$

Autrement dit, c'est la configuration dont la seule cellule à 1 est celle de coordonnée (i,j) d'où $\sigma_{(i,j)} = \tau_{(i,j)}(\sigma_{(0,0)})$.

Soit σ_0 une configuration initiale finie.

On peut écrire σ_0 comme une somme finie de configuration de la forme $\sigma_{(i,j)}$. On a

$$\sigma_0 = \bigoplus_{\substack{(x,y)\in\mathbb{Z}^2 \\ \sigma_0(x,y)=1}} \sigma_{(x,y)}$$

qui est en réalité une somme finie.

On peut donc appliquer le résultat précédent aux éléments de cette somme. Ainsi, à la $2^{i\text{ème}}$ génération, chacune de ces cellules est dupliquée 8 fois. De puis, si $2^i \leqslant M$, on ne peut pas trouver deux cellules du motif initial telles qu'elles ont un de leur huit copies qui se superpose. D'où le résultat. □

Cet automate réplique donc n'importe quel motif fini en entrée, c'est pourquoi il est aussi appelé automate réplicateur.

Chapitre 3

Les fonctions récursives

On introduit un nouveau formalisme pour caractériser un type de fonctions. Ce modèle de calcul n'est pas automatique comme les précédents (et d'autres qui suivent) mais purement mathématique. On définit un ensemble de fonctions par induction grâce à des schémas de construction de fonctions et des fonctions de bases. Les fonctions ainsi crées sont dites récursives. Obtenir le résultat consiste simplement à évaluer une fonction en un point. On voit donc qu'il n'y a pas de notion de pas de calcul, de mémoire ou d'état.

3.1 Les fonctions primitives récursives

3.1.1 Définition

Les fonctions manipulées ici sont des fonctions d'un sous-ensemble de $\mathbb{N}^k$ dans $\mathbb{N}^r$ pour $(r,k) \in \mathbb{N}^2$.

Définition 12 (Identité).

On appelle identité la fonction Id : $(n_1,\dots,n_p) \in \mathbb{N}^p \mapsto (n_1,\dots,n_p) \in \mathbb{N}^p$.

Définition 13 (Fonction nulle).

Pour tout entier $k \geqslant 0$, on appelle fonction nulle la fonction 0 : $(n_1,\dots,n_k) \in \mathbb{N}^k \mapsto 0 \in \mathbb{N}$.

Définition 14 (Projections).

Pour des entiers k et i tels que $0 \leqslant i \leqslant k$, la i-ème projection $p_{i,k}$ est définie par $p_{i,k} : (n_1, \ldots, n_k) \in \mathbb{N}^k \mapsto n_i \in \mathbb{N}$.

Définition 15 (Duplications).

Pour tout entier $r \geqslant 0$, la fonction de duplication d_r est définie par $d_r : n \in \mathbb{N} \mapsto (n, \ldots, n) \in \mathbb{N}^r$ où l'entier n est répété r fois.

Définition 16 (Successeur).

La fonction successeur succ est définie par $succ : n \in \mathbb{N} \mapsto n + 1 \in \mathbb{N}$.

Définition 17 (Composition de fonction).

On définit la composition.
Soit $p \in \mathbb{N}$ et soit $(k, r, k_1, \ldots, k_p) \in \mathbb{N}^{p+2}$. Soit p fonctions $(f_1, \ldots, f_p)$, où chaque fonction f_i pour $1 \leqslant i \leqslant p$ est une fonction de $\mathbb{N}^k$ dans $\mathbb{N}^{k_i}$, et soit g une fonction de $\mathbb{N}^{k_1+\cdots+k_p}$ dans $\mathbb{N}^r$, alors la composée $g(f_1, \ldots, f_p)$ est la fonction de $\mathbb{N}^k$ dans $\mathbb{N}^r$:

$$\begin{aligned} g(f_1, \ldots, f_p) : \mathbb{N}^k &\to \mathbb{N}^r \\ n &\mapsto g(f_1(n), \ldots, f_p(n)) \end{aligned}$$

Définition 18 (Construction récursive de fonction).

On définit une fonction récursivement.
Soit $(k, r) \in \mathbb{N}^2$, f une fonction de $\mathbb{N}^k$ dans $\mathbb{N}^r$ et g une fonction de $\mathbb{N}^{k+r+1}$ dans $\mathbb{N}^r$. La fonction $h = \mathrm{Rec}(f, g)$ est la fonction de $\mathbb{N}^{k+1}$ dans $\mathbb{N}^r$ définie pour tout $n \in \mathbb{N}$ et pour tout $m \in \mathbb{N}^k$ par :

$$\begin{aligned} h(0, m) &= f(m) \\ h(n+1, m) &= g(n, h(n, m), m) \end{aligned}$$

Définition 19 (Fonctions primitives récursives [Rig09]).

La famille des fonctions primitives récursives est la plus petite famille de fonctions qui vérifie les conditions suivantes :

— La fonction nulle, les projections, les fonctions de duplications et la fonction successeur sont primitives récursives.

— La famille des fonctions primitives récursives est close pour les compositions et la construction récursive de fonctions.

On remarque que toute fonction primitive récursive est définie en tout point. Cet argument est suffisant pour affirmer que toutes les fonctions calculables ne sont pas primitives récursives puisqu'il existe des fonctions calculables qui ne sont pas définies en certains points.

Notation 3.

On note $\mathcal{PR}$, la classe des fonctions primitives récursives. De plus on note $\mathcal{PR}\left(\mathbb{N}^m, \mathbb{N}^n\right)$, la classe des fonctions primitives récursives de $\mathbb{N}^m$ dans $\mathbb{N}^n$. En d'autres termes $\mathcal{PR}\left(\mathbb{N}^m, \mathbb{N}^n\right) = \mathcal{PR} \cap \left(\mathbb{N}^n\right)^{\left(\mathbb{N}^m\right)}$.

Les fonctions primitives récursives forment donc un ensemble assez large de fonctions. On va donner des exemples permettant de se rendre compte de leur expressivité, bien qu'il y ait des limitations qui nécessitent l'introduction des fonctions récursives.

3.1.2 Exemples

Exemple 4 (Prédécesseur).

La fonction prédécesseur peut être définie par

$$\begin{aligned} \text{pred}(0) &= 0 \\ \text{pred}(x+1) &= x \end{aligned}$$

On prend alors pour f la fonction nulle et pour g, la projection sur la première composante.

Exemple 5 (Somme).

La fonction somme de deux entiers est une fonction primitive récursive. Elle peut être définie par

$$\begin{aligned} \text{sum}\,(0, m) &= m \\ \text{sum}\,(n+1, m) &= \text{succ}\,(\text{sum}\,(n, m)) \end{aligned}$$

Exemple 6 (Produit).

La fonction produit peut être construite de la manière suivante :

$$\begin{aligned} \text{prod}\,(0, m) &= 0 \\ \text{prod}\,(n+1, m) &= \text{sum}\,(\text{prod}\,(n, m)\,, m) \end{aligned}$$

Elle est donc également primitive récursive.

On voit qu'on peut itérer de la même façon le produit pour avoir l'exponentiation et ainsi de suite. On définit donc une suite dont chacune est strictement plus croissante que la précédente. On peut toutefois montrer que la croissance de telles fonctions ne peut pas être arbitrairement grande.

Une hiérarchie des fonctions primitives récursives est basée sur cette suite de fonctions, la hiérarchie de GRZEGORCKYK. Elle sera étudié en détail par la suite.

Exemple 7.

La fonction $\mathit{isZero} : n \mapsto [n = 0]$ est primitive récursive. En effet :

$$\begin{aligned} \text{isZero} : \mathbb{N} &\to \mathbb{N} \\ n &\mapsto \begin{cases} 1 & \text{si } n = 0, \\ 0 & \text{sinon.} \end{cases} \end{aligned}$$

Ce qui est trivial avec une construction récursive.

On va maintenant donner des propriétés permettant de faire les opérations arithmétiques raisonnables sur les fonctions primitives récursives.

3.1.3 Propriétés

Définition 20 (Somme bornée).

On définit la somme bornée.
Soit $(k, r) \in \mathbb{N}^2$ et f une fonction de $\mathbb{N}^{k+1}$ dans $\mathbb{N}$. On définit alors la fonction $SumB\,(f)$ de $\mathbb{N}^{k+1}$ dans $\mathbb{N}$ de la manière suivante :

$$\begin{aligned} \text{SumB}\,(f) : \mathbb{N}^{k+1} &\to \mathbb{N} \\ (n, m) &\mapsto \sum_{i=0}^{n} f(i, m) \end{aligned}$$

Proposition 9.

$\mathcal{PR}$ est stable par somme bornée.

Démonstration. Soit $k \in \mathbb{N}$ et $f \in \mathcal{PR}$ f une fonction de $\mathbb{N}^{k+1}$ dans $\mathbb{N}$. On pose $g = \text{SumB}\,(f)$.

On a

$$\forall m \in \mathbb{N}^k, g(0, m) = f(0, m)$$

et

$$\forall (n, m) \in \mathbb{N} \times \mathbb{N}^k, g(n+1, m) = f(n+1, m) + g(n, m)$$

D'où

$$\forall (n, m) \in \mathbb{N} \times \mathbb{N}^k, g(n+1, m) = s(n, g(n, m), m)$$

où

$$\begin{aligned} s : \mathbb{N}^{k+2} &\to \mathbb{N} \\ (n, t, m) &\mapsto t + f(succ(n), m) \end{aligned}$$

On a alors

$$g = \text{Rec}\,(m \mapsto f(0, m), s)$$

ce qui est bien une construction récursive de fonction à partir de fonctions primitives récursives.

Ainsi, la somme bornée stabilise $\mathcal{PR}$. □

Définition 21 (Produit borné).

On définit le produit borné.
Soit $(k, r) \in \mathbb{N}^2$ et f une fonction de $\mathbb{N}^{k+1}$ dans $\mathbb{N}$. On définit alors la fonction ProdB (f) de $\mathbb{N}^{k+1}$ dans $\mathbb{N}$ de la manière suivante :

$$\begin{aligned} \text{ProdB}\,(f) : \mathbb{N}^{k+1} &\to \mathbb{N} \\ (n, m) &\mapsto \prod_{i=0}^{n} f(i, m) \end{aligned}$$

Proposition 10.

$\mathcal{PR}$ est stable par produit borné.

Démonstration. Soit $k \in \mathbb{N}$ et $f \in \mathcal{PR}$ f une fonction de $\mathbb{N}^{k+1}$ dans $\mathbb{N}$. On pose $g = \text{ProdB}\,(f)$.

On a

$$\forall m \in \mathbb{N}^k, g(0, m) = f(0, m)$$

et

$$\forall (n, m) \in \mathbb{N} \times \mathbb{N}^k, g(n+1, m) = f(n+1, m) \cdot g(n, m)$$

D'où

$$\forall (n, m) \in \mathbb{N} \times \mathbb{N}^k, g(n+1, m) = p(n, g(n, m), m)$$

où

$$\begin{aligned} p : \mathbb{N}^{k+2} &\to \mathbb{N} \\ (n, t, m) &\mapsto t \cdot f(succ(n), m) \end{aligned}$$

On a alors

$$g = \text{Rec}\,(m \mapsto f(0, m), p)$$

ce qui est bien une construction récursive de fonction à partir de fonctions primitives récursives.

Ainsi, le produit borné stabilise $\mathcal{PR}$. □

Définition 22 (Minimisation bornée).

On définit la minimisation bornée.
Soit $(k,r) \in \mathbb{N}^2$ et f une fonction de $\mathbb{N}^{k+1}$ dans $\mathbb{N}^r$. On définit alors la fonction MinB (f) de $\mathbb{N}^{k+1}$ dans $\mathbb{N}$ de la manière suivante :

$$\begin{aligned}\text{MinB}\,(f) : \mathbb{N}^{k+1} &\to \mathbb{N} \\ (n,m) &\mapsto \begin{cases} \min\{i \leqslant n \mid f\,(i,m) = d_r\,(0)\} & \text{s'il existe un tel } i \\ n+1 & \text{sinon.} \end{cases}\end{aligned}$$

Proposition 11.

$\mathcal{PR}$ est stable par minimisation bornée.

Démonstration. Soit $(k,r) \in \mathbb{N}^2$ et f une fonction de $\mathbb{N}^{k+1}$ dans $\mathbb{N}^r$.

$$\forall (n,m) \in \mathbb{N} \times \mathbb{N}^k, \text{MinB}\,(f)(n,m) = \sum_{k=0}^{n} \prod_{i=0}^{k} isZero(i,m)$$

Donc $\mathcal{PR}$ est stabilisé par la minimisation bornée. □

Définition 23.

La suite d'hyperopérateurs est la suite de fonctions de $\mathbb{N}^2$ à valeur dans $\mathbb{N}$ définie par

$$H_n(a,b) = \begin{cases} b+1 & \text{si } n = 0, \\ a & \text{si } n = 1 \wedge b = 0, \\ 0 & \text{si } n = 2 \wedge b = 0, \\ 1 & \text{si } n \geqslant 3 \wedge b = 0, \\ H_{n-1}(a, H_n(a, b-1)) & \text{sinon.} \end{cases}$$

Cette suite d'opérateur à un rôle centrale dans le hiérarchie de GRZEGORCZYK qu'on verra plus loin. Chacune de ces fonctions est simplement l'itérée de la précédente. Ainsi, H_0 est simplement le successeur du second argument, H_1 est la somme, H_2 est le produit, H_3 la puissance etc.

Notation 4 (Notation des flèches de KNUTH)**.**

Soit $(a, b, n) \in \mathbb{N}^3$ avec $n \geqslant 2$. $H_n(a, b)$ est noté $a \uparrow^{n-2} b$.

Parfois, on utilise abusivement la notation pour des entiers $n < 2$. On donne alors comme sens à $\uparrow^{-2}$ équivalent à H_0 et à $\uparrow^{-1}$, H_1. La notation étant parfois plus agréable que celle des hyperopérateurs.

Proposition 12.

$$\forall n \in \mathbb{N}, H_n \in \mathcal{PR}$$

Démonstration. Évident grâce à la définition. □

Malgré une définition fort peu intuitive, il existe une motivation aux fonctions primitives récursives. On peut en effet interpréter une fonction primitive récursive comme un programme constitué de boucles `for`. Aussi tout programme de ce genre termine toujours ce qui est cohérent avec le fait qu'une fonction primitive récursive est définie partout.

3.2 Les prédicats primitifs récursifs

Pour tout prédicat sur $\mathbb{N}^k$, il est naturel d'associer sa fonction caractéristique. Celle ci est en fait la fonction caractéristique de l'ensemble sur lequel le prédicat est vrai. Cela donne un sens à la notion de prédicat primitif récursif.

Définition 24.

Soit $k \in \mathbb{N}^*$ et P un prédicat sur $\mathbb{N}^k$. On dit que P est primitif récursif si sa fonction caractéristique

$$\begin{aligned} \mathbb{N}^k &\to \mathbb{N} \\ (x_1, \ldots, x_k) &\mapsto [P(x_1, \ldots, x_k)] \end{aligned}$$

est primitive récursive.

Proposition 13.

Soit P et Q deux prédicats primitifs récursifs. Les prédicats $\neg P$, $P \vee Q$, $P \wedge Q$, $P \Rightarrow Q$ et $P \Leftrightarrow Q$ sont primitifs récursifs.

Proposition 14 (Quantification bornée).

Soit $k \in \mathbb{N}^*$ et P un prédicat sur $\mathbb{N}^{k+1}$. Pour tout $n \in \mathbb{N}$, les prédicats sur $\mathbb{N}^k$

$$\begin{aligned} &\forall i \leqslant n, P(i, m) \\ &\exists i \leqslant n : P(i, m) \end{aligned}$$

sont primitifs récursifs.

Démonstration. Les indicatrices sont en effet respectivement

$$(n, m) \mapsto \prod_{i=0}^{n} f(i, m)$$

et

$$(n, m) \mapsto 1 - \prod_{i=0}^{n} (1 - f(i, m))$$

où f est la fonction indicatrice de P. □

Proposition 15 (Définition par disjonction de cas).

Soit $(k, r, n) \in (\mathbb{N}^*)^3$. Soit $P_1, \ldots, P_n$ des prédicats primitifs récursifs sur $\mathbb{N}^k$, $p_1, \ldots, p_n$ leur fonctions indicatrices respectives et $g_1, \ldots, g_{n+1}$ des fonctions primitives récursives de $\mathbb{N}^k$ dans $\mathbb{N}^r$. La fonction

$$\begin{aligned} &\mathbb{N}^k \to \mathbb{N}^r \\ &m \mapsto \begin{cases} g_1(m) & \text{si } P_1(m), \\ g_2(m) & \text{sinon si } P_2(m), \\ \quad\vdots & \\ g_n(m) & \text{sinon si } P_n(m), \\ g_{n+1}(m) & \text{sinon.} \end{cases} \end{aligned}$$

est primitive récursive.

Démonstration. Le but est d'expliciter cette disjonction de cas sans faire appel à des "sinon" qui ordonne chronologiquement la vérification des prédicats. Pour cela, on trouve des prédicats qui se vérifient dans les mêmes conditions (qui dont donc exclusifs) et sans dépendances à la valeur de vérité d'un autre prédicat.

On a

$$\begin{aligned} &\mathbb{N}^k \to \mathbb{N}^r \\ &m \mapsto \begin{cases} g_1(m) & \text{si } P_1(m), \\ g_2(m) & \text{si } P_2(m) \wedge \neg P_1(m), \\ \quad\vdots & \\ g_n(m) & \text{si } P_n(m) \wedge \bigwedge_{i=1}^{n-1} \neg P_i(m), \\ g_{n+1}(m) & \text{si } \bigwedge_{i=1}^{n} \neg P_i(m). \end{cases} \end{aligned}$$

D'où

$$\forall m \in \mathbb{N}^k, f(m) = \sum_{i=1}^{n} \left(g_i(m) \cdot p_i(m) \cdot \prod_{j=0}^{i-1} (1 - p_j(m)) \right)$$

□

Définition 25 (Arithmétique de PRESBURGER)**.**

Les prédicats de l'arithmétique de PRESBURGER sont :

- $\sum_{i=1}^{n} a_i x_i \geqslant k$,
- $\sum_{i=1}^{n} a_i x_i \equiv b[k]$,
- une combinaison booléenne de prédicats de l'arithmétique de PRESBURGER.

où $n \in \mathbb{N}$ $(a_i) \in \mathbb{Z}^n$, $(b, k) \in \mathbb{Z}^2$.
Les prédicats de l'arithmétique de PRESBURGER sont des prédicats sur $\mathbb{Z}^n$.

Les prédicats de l'arithmétique de PRESBURGER permettent de décrire exactement les ensembles semi-linéaires. C'est ce qui explique son importance en calculabilité.

Proposition 16.

Les prédicats de l'arithmétique de PRESBURGER sont primitifs récursifs.

Démonstration. Il suffit de montrer que les deux schémas de prédicats sont primitifs récursifs. Pour cela, il faut justifier que le modulo et les comparaisons avec $=$ et $\leqslant$ sont primitifs récursifs.

Tester l'inégalité large revient à faire la différence (dans $\mathbb{N}$) et regarder si elle est nulle. Tester l'égalité consiste à tester si la somme des deux différences est nulle.

Quant au modulo, il peut évidement se définit par cas et est donc primitif récursif. □

Proposition 17.

Les opérations arithmétiques de base sont primitives récursives. Plus exactement, l'addition, la multiplication, l'exponentiation, le factoriel, le prédécesseur, la soustraction dans $\mathbb{N}$, le minimum, le maximum, la valeur absolue, la fonction signe, la relation de divisibilité, le modulo, l'égalité, la comparaison, le test de primalité, la liste des nombres premiers, la décomposition en facteur premier et le logarithme sont primitifs récursifs.

3.3 Les fonctions récursives

Définition 26 (Minimisation).

On définit la minimisation.
Soit $(k, r) \in \mathbb{N}^2$ et f une fonction de $\mathbb{N}^{k+1}$ dans $\mathbb{N}^r$. On définit alors la fonction Min (f) de $\mathbb{N}^k$ dans $\mathbb{N}$ de la manière suivante :

$$\begin{aligned} \text{Min}\,(f) : \mathbb{N}^k &\to \mathbb{N} \\ m &\mapsto \min\{n \in \mathbb{N} \mid f\,(n, m) = d_r\,(0)\} \end{aligned}$$

Min (f) (m) peut donc éventuellement ne pas être définie.

Un résultat indéfini sera souvent noté $\perp$.

Définition 27 (Fonctions récursives [Car08b][Rig09]).

La famille des fonctions récursives est la plus petite famille de fonctions qui vérifie les conditions suivantes :

- Les fonctions primitives récursives sont récursives.
- La famille des fonctions récursives est close pour la composition, la construction récursive de fonctions et la construction de fonctions par minimisation.

On a ainsi la première différence triviale avec les fonctions primitives récursives qui sont définies partout.

Notation 5.

On note $\mathcal{R}$ l'ensemble des fonctions récursives.

Proposition 18.

$\mathcal{PR} \subsetneq \mathcal{R}$

Démonstration. Il suffit pour cela de construire une fonction qui ne soit pas définie en tout point. Par exemple la fonction $r = \text{Min}\,((m, n) \mapsto 1)$ d'une variable naturelle. On a $\forall m \in \mathbb{N}, r(m) = \perp$. □

Les fonctions récursives ne définissent pas exactement un modèle de calcul, mais une classe de fonctions. L'interprétation est des plus simples : le résultat est simplement l'évaluation mathématique de la fonction considérée. On verra par la suite l'intérêt de cette classe de fonctions par rapports aux modèles de calcul qu'on rencontre.

3.4 Exemples

On va développer l'exemple de la fonction d'ACKERMANN [Rig07].

Exemple 8 (La fonction d'ACKERMANN).

L'intérêt de cette fonction est d'expliciter une fonction qui est calculable, qui appartient donc aux fonctions récursives mais qui n'est pas primitive récursive.
On la définit sur $\mathbb{N}^2$ de la façon suivante :

$$\mathcal{A}(m,n) = \begin{cases} n+1 & \text{si } m = 0 \\ \mathcal{A}(m-1,1) & \text{si } m > 0 \text{ et } n = 0 \\ \mathcal{A}(m-1,\mathcal{A}(m,n-1)) & \text{si } m > 0 \text{ et } n > 0. \end{cases}$$

On montre facilement par induction que cette fonction est bien définie.
On note $\mathcal{A}_m = (n \in \mathbb{N} \mapsto \mathcal{A}(m,n))$.

Lemme 2.

$$\forall i \in \mathbb{N} \setminus \{0,1\}, H_i(2,1) = 2$$

Démonstration. Soit $i \in \mathbb{N} \setminus \{0,1\}$,
$H_i(2,1) = H_{i-1}(2, H_i(2,0))$.

- Si $i = 2$: $H_1(2, H_2(2,0)) = H_1(2,0) = 2$
- Si $i = 3$: $H_2(2, H_3(2,0)) = H_2(2,1) = 2$, en se ramenant au cas précédent.
- Si $i > 3$: $H_{i-1}(2, H_i(2,0)) = H_{i-1}(2,1) = 2$, par une récurrence évidente sur i.

□

Lemme 3.

$$\forall n \in \mathbb{N}^*, H_i(2,2) = 4$$

Démonstration. Soit $i \in \mathbb{N}^*$,
$H_i(2,2) = H_{i-1}(2, H_i(2,1)) = H_{i-1}(2, H_{i-1}(2, H_i(2,0)))$.

— Si $i = 1 : H_0(2, H_0(2, H_1(2,0))) = H_0(2, H_0(2,2)) = H_0(2,3) = 4$
— Si $i = 2 : H_1(2, H_1(2, H_2(2,0))) = H_1(2, H_1(2,0)) = H_1(2,2) = 4$, en se ramenant au cas précédent.
— Si $i \geqslant 3 : H_{i-1}(2, H_{i-1}(2, H_i(2,0))) = H_{i-1}(2,2) = 4$, par une récurrence évidente sur i.

□

Proposition 19.

$$\forall (m,n) \in \mathbb{N}^2, \mathcal{A}(m,n) = H_m(2, n+3) - 3$$

Démonstration. On prouve cette égalité par induction sur $\mathbb{N}^2$.

Si $m = 0, \forall n \in \mathbb{N}, H_0(2,(n+3)) - 3 = (n+3) + 1 - 3 = n + 1 = \mathcal{A}(m,n)$

Si $m > 0$ et $n = 0$, $H_m(2,3) - 3 = H_{m-1}(2, H_m(2,2)) - 3 = H_{m-1}(2,4) - 3 = \mathcal{A}(m-1,1)$

Sinon,

$$\begin{aligned} H_m(2,n+3) - 3 &= H_{m-1}(2, H_m(2,(n-1)+3)) \\ &= \mathcal{A}(m-1, \mathcal{A}(m,n-1)) \end{aligned}$$

□

Remarque 2.

Pour tout $m \in \mathbb{N}$, $\mathcal{A}_m$ est primitive récursive.
On procède par récurrence sur m.
Cas de base immédiat : $\mathcal{A}(0,n) = n+1$.
Supposons $\mathcal{A}_m$ primitive récursive et montrons que H_{m+1} l'est.
On dispose du schéma de récursion primitive suivant

$$\begin{cases} \mathcal{A}_{m+1}(0) &= \mathcal{A}_m(1) \\ \mathcal{A}_{m+1}(n+1) &= \mathcal{A}_m(\mathcal{A}_{m+1}(n)) \end{cases}$$

On remarque que $\forall m \in \mathbb{N}, \mathcal{A}_m \in \mathcal{PR}$ n'implique pas $\mathcal{A} \in \mathcal{PR}$.

Lemme 4.

$$\forall (m,n) \in \mathbb{N}^2, \mathcal{A}_m(n) > n$$

Démonstration. On procède par une double récurrence.

Par récurrence sur m.
Pour $m = 0$ et pour tout n, $\mathcal{A}_0(n) = n + 1 > n$.
Supposons que, pour tout n, $\mathcal{A}_m(n) > n$ et vérifions que, pour tout n, $\mathcal{A}_{m+1}(n) > n$.

On procède par récurrence sur n. Si $n = 0$, alors grâce à l'hypothèse de récurrence sur m, on a bien $\mathcal{A}_{m+1}(0) = \mathcal{A}_m(1) > 1 > 0$.
Supposons à présent que $\mathcal{A}_{m+1}(n) > n$ et vérifions que $\mathcal{A}_{m+1}(n+1) > n + 1$. Il vient en utilisant successivement les hypothèses de récurrence sur m et sur n : $\mathcal{A}_{m+1}(n+1) = \mathcal{A}_m(\mathcal{A}_{m+1}(n)) > \mathcal{A}_{m+1}(n) > n$ d'où le résultat annoncé, $\mathcal{A}_{m+1}(n+1) > n + 1$, puisque nous sommes en présence de deux inégalités strictes consécutives. □

Lemme 5 (Croissance stricte par rapport à la deuxième variable).

$$\forall (m,n) \in \mathbb{N}^2, \mathcal{A}_m(n+1) > \mathcal{A}_m(n)$$

Démonstration. Si $m = 0$, c'est immédiat puisque $\mathcal{A}_0(n) = n + 1$.
Si $m > 0$, alors en utilisant le lemme précédent, il vient
$\mathcal{A}_m(n+1) = \mathcal{A}_{m-1}(\mathcal{A}_m(n)) > \mathcal{A}_m(n)$. □

Lemme 6 (Croissance par rapport à la première variable).

$$\forall (m,n) \in \mathbb{N}^2, \mathcal{A}_{m+1}(n) \geqslant \mathcal{A}_m(n)$$

Démonstration. Pour $n = 0$, on a $\mathcal{A}_{m+1}(0) = \mathcal{A}_m(1) > \mathcal{A}_m(0)$ car $\mathcal{A}_m$ est une fonction strictement croissante (cf. lemme 5). Pour $n > 0$, au vu du lemme 4, $\mathcal{A}_{m+1}(n-1) \geqslant n$ et puisque $\mathcal{A}_m$ est une fonction strictement croissante, il vient $\mathcal{A}_m(\mathcal{A}_{m+1}(n-1)) \geqslant \mathcal{A}_m(n)$. En appliquant la définition de la fonction d'ACKERMANN, on en conclut que

$$\mathcal{A}_{m+1}(n) = \mathcal{A}_m(\mathcal{A}_{m+1}(n-1)) \geqslant \mathcal{A}_m(n)$$

□

On note ${\mathcal{A}_m}^k$ l'itéré $k^{\text{ème}}$ de $\mathcal{A}_m$. En particulier, ${\mathcal{A}_m}^0$ est la fonction identité.

Lemme 7.

Les fonctions ${\mathcal{A}_m}^k$ sont toutes strictement croissantes. De plus, pour tous $(m,n,k) \in \mathbb{N}^3$ on a

— ${\mathcal{A}_m}^{k+1}(n) > {\mathcal{A}_m}^k(n)$

— ${\mathcal{A}_m}^k(n) \geqslant n$

— $m \leqslant p \Rightarrow {\mathcal{A}_m}^k(n) \leqslant {\mathcal{A}_p}^k(n)$.

Démonstration. La preuve est immédiate et découle principalement du fait que $\mathcal{A}_m$ est une fonction strictement croissante. La dernière assertion découle du lemme 6. □

Définition 28.

Soit $(f,g) \in \mathbb{N}^{\mathbb{N}} \times \mathbb{N}^{(\mathbb{N}^p)}$. La fonction f domine g, ce que l'on notera $g \prec f$ si

$$\exists C \in \mathbb{N} : \forall (x_1,\ldots,x_p) \in \mathbb{N}^p, g(x_1,\ldots,x_p) \leqslant f(\max(x_1,\ldots,x_p,C))$$

Cette notion permet d'observer la croissance des fonctions en s'affranchissant des comportement autour de 0. On a ainsi un outil pour comparer les croissances ultimes.

Remarque 3.

Si $f \in \mathbb{N}^{\mathbb{N}}$ est une fonction strictement croissante, alors $g \prec f$ si, et seulement si, $g(x_1,\ldots,x_p) \leqslant f(\max(x_1,\ldots,x_p))$ sauf pour un nombre fini de points.

($\Rightarrow$) Le nombre de p-uplets $(x_1,\ldots,x_p) \in \mathbb{N}^p$ tels que $\max(x_1,\ldots,x_p) < C$ est majoré par C^p et est donc fini.

($\Leftarrow$) Soit $(a_1,\ldots,a_t)$ les points (en nombre fini) tels que $g(a_i) > f(\max(a_i))$. On pose $C = \max(g(a_1),\ldots,g(a_t))$. Puisque f est strictement croissante par hypothèse, $\exists D : \forall x, x \geqslant D \Rightarrow f(x) > C$. Donc, $\forall x \in \mathbb{N}^p, g(x) \leqslant f(\max(x,D))$ et $g \prec f$.

Notation 6.

Pour $m \geqslant 0$, on pose

$$\mathfrak{T}_m = \left\{ g \in \mathbb{N}^{(\mathbb{N}^p)} \;\middle|\; \exists k \in \mathbb{N} : g \prec {\mathcal{A}_m}^k \right\}$$

ie. l'ensemble des fonctions dominées par un itéré de la fonction $\mathcal{A}_m$ et

$$\mathfrak{T} = \bigcup_{m \in \mathbb{N}} \mathfrak{T}_m$$

Les fonctions primitives récursives de base appartiennent à $\mathfrak{T}_0$.

En outre, il est évident que si $(f, g) \in \mathbb{N}^{(\mathbb{N}^p)}$, si $g \in \mathfrak{T}_m$ et si $\forall \left(x_1, \ldots, x_p\right) \in \mathbb{N}^p, f\left(x_1, \ldots, x_p\right) \leqslant g\left(x_1, \ldots, x_p\right)$ alors f appartient aussi à $g \in \mathfrak{T}_m$.

Remarque 4.

Il est aisé de se convaincre que les fonctions
- $sup : \left(x_1, \ldots, x_p\right) \in \mathbb{N}^p \mapsto \max\left(x_1, \ldots, x_p\right)$
- $\Sigma_2 : (m, n) \in \mathbb{N}^2 \mapsto m + n$
- pour $k \in \mathbb{N}, n \in \mathbb{N} \mapsto kn$

appartiennent toutes à $\mathfrak{T}_2$.
Cette observation nous sera utile pour la suite.

Lemme 8.

Pour tout $m \geqslant 0$, l'ensemble $\mathfrak{T}_m$ est stable par composition. Par conséquent, $\mathfrak{T}$ l'est aussi.

Démonstration. Soit $(f_1, \ldots, f_n)$ des fonctions de $\mathbb{N}^{(\mathbb{N}^p)} \bigcap \mathfrak{T}_m$ et une fonction $g \in \mathbb{N}^{(\mathbb{N}^n)} \bigcap \mathfrak{T}_m$.

Par définition de l'ensemble $\mathfrak{T}_m$, il existe des n-uplets de constantes naturelles $(k, k_1, \ldots, k_n)$ et $(C, C_1, \ldots, C_n)$ telles que

$$\forall \left(y_1, \ldots, y_n\right) \in \mathbb{N}^n, g\left(y_1, \ldots, y_n\right) \leqslant {\mathcal{A}_m}^k \left(\max\left(y_1, \ldots, y_n, C\right)\right)$$

et pour tout $i \in [\![1, n]\!]$, on a

$$\forall (x_1, \ldots, x_p) \in \mathbb{N}^p, f_i (x_1, \ldots, x_p) \leqslant {\mathcal{A}_m}^{k_i} (\max (x_1, \ldots, x_p, C_i))$$

Posons $D = \max (C, C_1, \ldots, C_n)$ et $K = \max (k_1, \ldots, k_n)$, il vient

$$\begin{aligned} g(f_1(x_1, \ldots, x_p), \ldots, & f_n(x_1, \ldots, x_p)) \\ & \leqslant {\mathcal{A}_m}^k (\max (f_1 (x_1, \ldots, x_p), \ldots, f_n (x_1, \ldots, x_p), C)) \\ & \leqslant {\mathcal{A}_m}^k \left({\mathcal{A}_m}^K (\max (x_1, \ldots, x_p, D)) \right) \\ & \leqslant {\mathcal{A}_m}^{k+K} (\max (x_1, \ldots, x_p, D)) \end{aligned}$$

où, à l'avant-dernière ligne, on a utilisé le lemme 7. □

Lemme 9.

$$\forall (m, n, k) \in \mathbb{N}^3, {\mathcal{A}_m}^k (n) \leqslant \mathcal{A}_{m+1} (n + k)$$

Démonstration. On montre cela grâce à une récurrence sur k en remarquant que :

$$\begin{aligned} \mathcal{A}^{k+1} (m, n) &= \mathcal{A} \left(m, \left(\mathcal{A}^k (m, n) \right) \right) \\ &\leqslant \mathcal{A} (m, \mathcal{A} (m + 1, n + k)) \\ &\leqslant \mathcal{A} (m + 1, n + k + 1) \end{aligned}$$

□

Lemme 10.

Si $g \in \mathbb{N}^{(\mathbb{N}^p)}$ et $h \in \mathbb{N}^{\left(\mathbb{N}^{p+2}\right)}$ sont deux fonctions de $\mathfrak{T}_m$, alors la fonction $\mathrm{Rec}\,(g, h) \in \mathfrak{T}_{m+1}$. Par conséquent, $\mathfrak{T}$ est stable par récursion primitive.

Soit $(g, h) \in \mathbb{N}^{(\mathbb{N}^p)} \times \mathbb{N}^{\left(\mathbb{N}^{p+2}\right)}$ deux fonctions de $\mathfrak{T}_m$. On considère la fonction f définie par :

$$\left\{ \begin{array}{rcl} f (0, x_1, \ldots, x_p) & = & g (x_1, \ldots, x_p) \\ f (n + 1, x_1, \ldots, x_p) & = & h (n, f (n, x_1, \ldots, x_p), x_1, \ldots, x_p) \end{array} \right.$$

ie $f = \mathrm{Rec}\,(g, h)$.

Par définition de l'ensemble $\mathfrak{T}_m$, il existe des constantes $(A_1, A_2, k_1, k_2) \in \mathbb{N}^4$ telles que

$$\forall (x_1, \ldots, x_p) \in \mathbb{N}^p : g(x_1, \ldots, x_p) \leqslant {\mathcal{A}_m}^{k_1} (\max(x_1, \ldots, x_p, A_1))$$

et

$$\forall (x_1, \ldots, x_p, x_{p+1}, x_{p+2}) \in \mathbb{N}^{p+2},$$
$$h(x_1, \ldots, x_p, x_{p+1}, x_{p+2}) \leqslant {\mathcal{A}_m}^{k_2} (\max(x_1, \ldots, x_p, x_{p+1}, x_{p+2}, A_2))$$

On montre tout d'abord par récurrence sur n que

$$f(n, (x_1, \ldots, x_p)) \leqslant {\mathcal{A}_m}^{k_1+nk_2} (\max(x_1, \ldots, x_p, n, A_1, A_2)) \tag{3.1}$$

Le résultat est vrai pour $n = 0$. Supposons-le satisfait pour n et vérifions-le pour $n+1$. Il vient

$$f(n+1, (x_1, \ldots, x_p)) \leqslant {\mathcal{A}_m}^{k_2} (\max(x_1, \ldots, x_p, n, f(n, (x_1, \ldots, x_p)), A_2))$$

En appliquant l'hypothèse de récurrence, on trouve

$$\begin{aligned} f(n+1, x_1, \ldots, x_p) &\leqslant {\mathcal{A}_m}^{k_2} \left({\mathcal{A}_m}^{k_1+nk_2} (\max(x_1, \ldots, x_p, n, A_1, A_2)) \right) \\ &\leqslant {\mathcal{A}_m}^{k_1+(n+1)k_2} (\max(x_1, \ldots, x_p, n, A_1, A_2)) \\ &\leqslant {\mathcal{A}_m}^{k_1+(n+1)k_2} (\max(x_1, \ldots, x_p, n+1, A_1, A_2)) \end{aligned}$$

Nous pouvons à présent conclure. Par le lemme précédent appliqué à (3.1),

$$f(n+1, (x_1, \ldots, x_p)) \leqslant \mathcal{A}_{m+1} (\max(x_1, \ldots, x_p, n, A_1, A_2) + k_1 + (n+1) k_2)$$

$$f(n+1, (x_1, \ldots, x_p)) \leqslant \mathcal{A}_{m+1} (\max(x_1, \ldots, x_p, n, A_1, A_2) + k_1 + (n+1) k_2)$$

Le membre de droite est la composée d'une fonction appartenant à $\mathfrak{T}_{m+1}$ (à savoir $\mathcal{A}_{m+1}$) avec une fonction de $\mathfrak{T}_2$. Pour $m > 1$, elle appartient donc à $\mathfrak{T}_{m+1}$ au vu du lemme 8. Pour $m \in \{0, 1\}$, elle appartient à $\mathfrak{T}_2$.

D'où la même conclusion pour f.

Corollaire 2.

On a $\mathcal{PR} \subseteq \mathfrak{T}$.

Théorème 2.

La fonction d'ACKERMANN n'est pas primitive récursive.

$$\mathcal{A} \notin \mathcal{PR}$$

Démonstration. On procède par l'absurde.

Supposons $\mathcal{A}$ récursive primitive.

La fonction $f : n \mapsto \mathcal{A}(n, 2n)$ est alors elle aussi récursive primitive. Au vu de la proposition précédente ($f \in \mathfrak{T}$), il existe des constantes C, m et k telles que $\forall n \in \mathbb{N}, \Big(n > C \Rightarrow \mathcal{A}(n, 2n) \leqslant \mathcal{A}^k(m, n)\Big)$. Ainsi, en appliquant le lemme 9, $\forall n \in \mathbb{N}, \Big(n > C \Rightarrow \mathcal{A}(n, 2n) \leqslant \mathcal{A}^k(m, n) \leqslant \mathcal{A}(m+1, n+k)\Big)$.

D'autre part, si $n > \max(C, k, m+1)$, alors

$$\mathcal{A}(m+1, n+k) < \mathcal{A}(m+1, 2n) < \mathcal{A}_m(2n) = \mathcal{A}(n, 2n)$$

d'où une contradiction. □

On a donc explicité une fonction calculable et non récursive primitive, d'où l'intérêt de définir les fonctions récursives en plus des fonctions récursives primitives.

La fonction d'ACKERMANN présente un autre intérêt :

$$\forall f \in \mathcal{PR}(\mathbb{N}^p, \mathbb{N}), \exists n \in \mathbb{N} : \forall i \in \mathbb{N}, i \geqslant n \Rightarrow f \prec \mathcal{A}_n$$

Autrement dit, elle majore strictement toutes les fonctions récursives primitives (cf. corolaire 6). D'autre part, $\mathcal{A} \in \mathcal{R}$.

Chapitre 4

Le λ-calcul

Le λ-calcul est un modèle d'un genre très particulier. Il ressemble à un modèle mathématique car il permet de définir un ensemble de fonctions, mais en même temps, il est partiellement automatique puisqu'on peut tracer son exécution pas à pas. Les données manipulées ici sont des arbres très simples qui définissent des fonctions très générales. Il existe des règles de calculs qui permettent de transformer cet arbre, l'exécution de se modèle sera donc simplement une suite d'utilisation de ces règles de façon à obtenir un arbre, souvent plus simple. On peut donc trouver la liste des arbres intermédiaires lors du calcul.

4.1 Définition

On se donne un ensemble infini $\mathcal{V}$. On peut prendre par exemple $\Sigma^\star$ pour n'importe quel alphabet non vide Σ.

Définition 29 (λ-terme)**.**

On définit récursivement un λ-terme :
- les éléments de $\mathcal{V}$ sont appelées des variables sont des λ-termes.
- $u\,v$ est un λ-terme si u et v sont des λ-termes (applications)
- $\lambda x.u$ est un λ-terme si u est un λ-terme et x une variable (abstractions)

On peut donner une syntaxe sous forme BNF :

$$u, v ::= x \mid u\,v \mid \lambda x.u$$

Notation 7 (Règles de parenthésage)**.**

L'abstraction porte aussi loin que possible. Par exemple :

$$\lambda x.\lambda y.\lambda z.x\,y\,z = \lambda x.\,(\lambda y.\,(\lambda z.x\,y\,z))$$

L'application est associative à gauche. Par exemple :

$$x\,y\,z = (x\,y)\,z$$

Notation 8 (Curryfication)**.**

$$\lambda x.\lambda y.u = \lambda x\,y.u$$

Ici, ce n'est qu'une facilité, on n'exploitera pas le détail de la curryfication car c'est hors du propos de la calculabilité.

Pour mieux comprendre la suite, il est préférable d'avoir une interprétation intuitive des λ-termes, on peut en effet traduire un λ-terme en une fonction en mathématique classique.

Une variable du λ-calcul représente une variable de fonction.

L'application $u\,v$ peut être interprétée comme $u(v)$. On remarque que la fonction u prend en paramètre une fonction. Bien que ça peut être surprenant pour certains, ça ne constitue pas une difficulté en soi.

L'abstraction $\lambda x.e$ sera comprise comme $x \mapsto e$. Un tel terme correspond à l'introduction d'une fonction.

On illustre avec quelques exemples.

Exemple 9.

Le λ-terme $\lambda x.c$ est la fonction constante égale à c.

Exemple 10.

Le λ-terme $\lambda x.x$ est l'identité.

Exemple 11.

Le λ-terme $\lambda c.\lambda x.c$ est la fonction qui construit des fonctions constantes. En mathématiques, elle correspond à $f = c \mapsto (x \mapsto c)$.
Ainsi, pour c, $f(c)$ est une fonction constante, autrement dit, pour tout x, on a $f(c)(x) = c$.

4.2 Règles de calcul

Maintenant que nous avons une interprétation intuitive des λ-termes, on va pouvoir comprendre la logique des règles de calcul qui suivent.

4.2.1 α-équivalence

Tout d'abord, on définit une forme d'équivalence qui met en relation les termes qui représentent intrinsèquement la même chose. Si $I_x := \lambda x.x$ représente l'identité, il est assez évident que $I_y := \lambda y.y$ représente la même fonction. Cependant, on ne peut pas a priori écrire que $I_x = I_y$, à cause de la définition inductive de l'égalité. Une observation intéressante, c'est qu'on peut toutefois noter $\forall c, I_x\, c = I_y\, c$. Cependant, une telle caractérisation compare les comportements de I_x et de I_y, et non leur structure. Par exemple, on a $\forall c, (\lambda x.x)\, c = c = (\lambda x.(\lambda t.t)\, x)\, c$. Seulement, $\lambda x.(\lambda t.t)\, x$ a une structure intrinsèquement différente de celle de I_x.

L'équivalence qu'on veut mettre en avant, est celle qui utilise le fait que renommer une variable dans une abstraction ne change pas le sens du λ-terme. De la même façon que $x \mapsto f(x) = y \mapsto f(y)$ ou $\int f(t)\mathrm{d}t = \int f(x)\mathrm{d}x$.

Définition 30 (Variables libres).

Les variables libres d'un λ-terme sont définies par induction de la façon suivante :

- $\text{freeVars}\,(x) = \{x\}$
- $\text{freeVars}\,(e_1\, e_2) = \text{freeVars}\,(e_1) \cup \text{freeVars}\,(e_2)$
- $\text{freeVars}\,(\lambda x.e) = \text{freeVars}\,(e) \setminus \{x\}$

Une variable non libre est dite liée.

Exemple 12.

$$\text{freeVars}\,(\lambda x\, z.x\, y) = \{y\}$$
$$\text{freeVars}\,(u\, v) = \{u, v\}$$

Les variables libres sont celles qui ne sont pas liées par une abstraction. Leur valeur dépend donc du contexte.

Définition 31 (Terme clos).

Un λ-terme t est clos si $\text{freeVars}(t) = \emptyset$.

Exemple 13.

$$\lambda x\, y.x$$

est clos mais

$$x\, y$$

ne l'est pas.

Définition 32 (Substitution).

La substitution d'un terme M à une variable x dans le terme e, notée $[M/x]\, e$, est définie pas induction par :

- $[M/x]\, x = M$
- $[M/x]\, y = y$ pour $x \neq y$
- $[M/x]\, (e_1\, e_2) = ([M/x]\, e_1)\, ([M/x]\, e_2)$
- $[M/x]\, (\lambda x.e) = \lambda x.e$
- $[M/x]\, (\lambda y.e) = \lambda y.\, [M/x]\, e$ pour $x \neq y$ et $y \notin \text{freeVars}(M)$
- $[M/x]\, (\lambda y.e) = \lambda z.\, [M/x]\, ([z/y]\, e)$ pour $z \notin \text{freeVars}(e) \cup \text{freeVars}(M))$ et $z \neq x$ et $y \neq x$

Exemple 14.

$$[\lambda x.y/z]\, (\lambda z.z)\, (\lambda i.z)\, (\lambda y.y\, x) = (\lambda z.z)\, (\lambda i.\lambda x.y)\, (\lambda u.u\, x)$$

La substitution $[M/x]\, t$ remplace chaque occurrence de x par M dans t. On lit « M que remplace x dans t ». Certains auteurs préfèrent écrire $t[M/x]$. Bien entendu, ça ne change rien d'autre que la notation.

On remarque que dans le dernier cas, on a besoin d'une nouvelle variable z, dite variable fraiche. Il est toujours possible d'en trouver une, puisque l'ensemble des variables est infini alors qu'un λ-terme est toujours fini.

Définition 33.

Une relation $\mathcal{R}$ sur les λ-terme est dite stable par construction si pour tout λ-termes u, v, u', v' et variable x, on a

$$u \mathcal{R} u' \Rightarrow \lambda x.u \mathcal{R} \lambda x.u'$$
$$u \mathcal{R} u' \Rightarrow u\, v \mathcal{R} u'\, v$$
$$v \mathcal{R} v' \Rightarrow u\, v \mathcal{R} u\, v'$$

Définition 34 (α-équivalence).

L'alpha équivalence noté $\leftrightarrow_\alpha$ (ou $\leftrightarrow_\alpha$) est l'intersection des relations d'équivalences stables par construction telles que

$$\lambda x.e \leftrightarrow_\alpha \lambda y.\, [y/x]\, e$$

pour y n'apparaissant pas dans e.

Exemple 15.

$$\lambda x.u\, x \leftrightarrow_\alpha \lambda y.u\, y$$

En fait, on peut même dire que c'est la plus petite relation qui vérifie la condition puisque c'est la clôture réflexive, symétrique et transitive de $\lambda x.e \to \lambda y.\, [y/x]\, e$ où y n'apparait pas dans e.

L'égalité de λ-termes se fait toujours à α-équivalence près. On identifie alors des termes α-équivalents. Par conséquent, on se permettra d'écrire $I_x = I_y$, sans même faire allusion à l'α-équivalence.

4.2.2 β-équivalence

Maintenant, on va définir l'évaluation. En mathématique, quand on a $(x \mapsto f(x))(e)$, on peut évaluer en $f(e)$. Pour cela, on remplace toutes les occurrences de la variable de fonction (dans l'exemple x) par l'argument (dans l'exemple e). De la même façon, on va définir $(\lambda x.f)\, e$. En remplaçant dans f toutes les occurrences de x par e.

Définition 35 (β-réduction).

On appelle β-réduction la plus petite relation stable par construction qui vérifie

$$(\lambda x.e)\, M \to_\beta [M/x]\, e$$

Exemple 16.

$$\lambda t.(\lambda u.v\, u)\, (\lambda x.y) \to_\beta \lambda t.(v\, (\lambda x.y))$$

Cette règle de calcul constitue la principale méthode d'évaluation d'un λ-terme. Grâce à elle, un λ-terme n'est pas qu'un arbre, mais prend le sens de fonction annoncée précédemment.

Remarque 5.

Un (sous-)terme de la forme $(\lambda x.e)\, M$ est appelé redex pour *reductible expression*, en effet ce sont les termes qu'on peut réduire.

Définition 36 (β-équivalence).

On note $=_\beta$ (ou$\leftrightarrow_\beta$) la fermeture transitive, symétrique et réflexive de $\to_\beta$, on l'appelle la β-équivalence.

Ainsi, on a tout ce qu'il faut pour faire du calcul avec des λ-terme. Tout ce qu'il reste à faire, c'est de se donner des encodages canoniques de ce qu'on veut calculer (entiers, booléens) après avoir défini un dernier type d'équivalence.

4.2.3 η-équivalence

Définition 37 (η-équivalence).

On appelle η-équivalence la plus petite relation d'équivalence $\leftrightarrow_\eta$ telle que

$$\lambda x.e\, x \leftrightarrow_\eta e$$

C'est l'égalité mathématique entre $x \mapsto f(x)$ et f, Certains connaissent aussi cette équivalence dans certains langages de programmations fonctionnels. Par exemple, cela revient à `function n->f n` et `f` en OCaml ou entre `\n -> f n` et `f` en Haskell.

4.3 Les stratégies d'évaluation, déterminisme et terminaison

Dans un λ-terme, tout ce qu'on peut faire, c'est de réduire un redex. Seulement, il peut y avoir plusieurs redex. Dans ce cas, on ne sait pas a priori lequel choisir. A priori, on ne peut pas conclure l'équivalence de toutes les réductions, ni même qu'elles terminent toutes ou aucune.

On peut remarquer dans un premier temps que certains termes ont des réductions qui ne termines pas. On donne un exemple.

Exemple 17.

On pose $\Delta := \lambda x.x\ x$ et $\Omega := \Delta\ \Delta$.
On a un seul redex donc une seule réduction possible. Et on a $\Omega \to_\beta \Omega$.
La réduction ne termine pas.

Un terme est dit normal (ou sous forme normale) s'il ne contient pas de redex. Un terme t est dit normalisable si il existe u tel que $t =_\beta u$. Et il est dit fortement normalisable si toutes les réductions mènent à une forme normale.

Théorème 3 (CHURCH-ROSSER).

Soit t et u deux λ-termes tels que $t =_\beta u$ il existe un λ-terme v tel que $t \to_\beta^* v$ et $u \to_\beta^* v$.

Démonstration. Avec les définitions données, le théorème est évident. □

Théorème 4 (Théorème de confluence).

Soit t, u_1 et u_2 des λ-termes.
Si $t \to_\beta^* u_1$ et $t \to_\beta^* u_2$, alors il existe un λ-terme v tel que $u_1 \to_\beta^* v$ et $u_2 \to_\beta^* v$.

Démonstration. On fait une preuve par induction sur les β-réductions. □

En somme, on peut dire que lorsqu'on choisit un redex, il est jamais trop tard pour corriger une erreur éventuelle.

Exemple 18.

Soit t le terme $(\lambda x.c)\ ((\lambda x.x\ x)\ (\lambda x.x\ x))$. En notant en bleu l'abstraction et en rouge son argument, on voit les deux redex

$$(\lambda x.c)\ ((\lambda x.x\ x)\ (\lambda x.x\ x))$$

et

$$(\lambda x.c)\ ((\lambda x.x\ x)\ (\lambda x.x\ x))$$

Si on réduit le premier redex, on trouve immédiatement c. Si on réduit le second, le terme ne change pas. On peut faire cette opération un nombre arbitraire de fois et à tout moment réduire l'autre redex et trouver c. Ainsi, on vérifie dans ce cas le théorème de confluence, mais on remarque aussi qu'il existe des termes normalisables qui ne sont pas fortement normalisables.

Naturellement, on peut chercher des stratégies d'évaluation qui termineraient sur tout terme normalisable. Une stratégie d'évaluation est un moyen de choisir le redex à réduire à chaque étape. Ceci introduit une contrainte dans l'ordre d'évaluation d'un lambda-terme. Jusqu'à présent, on pouvait avoir plusieurs suites de termes obtenus par β-réduction. Avec une stratégie d'évaluation, il n'y a plus de choix et il n'existe plus qu'une seule suite. On dit que le système est déterministe. Cette propriété s'observe ou se nie dans n'importe quel système à transition. Jusqu'à présent nous n'avons vu que des systèmes déterministes. Par exemple, étant donné une configuration, une machine de TURING n'évoluera que vers une unique autre configuration. Il existe une variante non déterministe (astucieusement nommée machine de TURING non déterministe) qui permet plusieurs transitions valables. Il faut alors redéfinir la notion de terminaison, de réussite d'un calcul...

Le λ-calcul est par essence non déterministe. Il est possible de fournir des stratégies d'évaluation pour rendre son évaluation déterministe. Par exemple, on peut systématiquement réduire le plus profondément possible, en choisissant d'abord le terme de droite des applications. On peut aussi utiliser la stratégie inverse : faire toutes les substitutions les plus proches de la racine d'abord.

Cependant, pour toute stratégie, il existe des termes normalisables dont la réduction ne terminera pas. En effet, on verra par la suite que l'existence d'une forme normale pour un λ-terme est indécidable par équivalence avec le problème de l'arrêt d'une machine de TURING.

Ainsi, aussi tentant soit il, nous ne donneront pas de stratégie d'évaluation et le λ-calcul sera le seul modèle non déterministe de cette partie.

4.4 Programmation en λ-calcul

Dans l'optique de prouver la TURING-équivalence du λ-calcul, on définit quelques structures permettant de construire des algorithmes en λ-calcul.

Il est en effet nécessaire de donner des encodages canoniques des différents types de données, puisque le λ-calcul ne fournit a priori rien de ressemblant aux entiers et autres valeurs mathématiques classiques.

4.4.1 Les entiers

Tout d'abord, on définit les entiers naturels. Cet encodage des entiers naturels est assez unanime. Le nombre n est représenté par la fonction qui prend une fonction en argument et calcul son itéré $n^{\text{ème}}$. On parle de fonctionnel d'itération.

Ainsi, on a

Notation 9.

$$\overline{n} = \lambda f\, x.f^n x$$

Par exemple :
— $\overline{0} = \lambda f\, x.x$
— $\overline{1} = \lambda f\, x.f\, x$
— $\overline{2} = \lambda f\, x.f\, (f\, x)$

On peut donc donner de façon assez directe la fonction succ qui calcul le successeur.

Notation 10.

$$\text{succ} = \lambda n\, f\, x.f\, ((n\, f)\, x)$$

On les appelle cette représentation « entier de CHURCH ». Cette définition est purement arbitraire et ne se base sur aucune propriété intuitive des entiers naturels.

On justifie la définition de succ.

Proposition 20.

$$\text{succ } \overline{n} =_\beta \overline{n+1}$$

Démonstration.

$$\begin{aligned}
\text{succ } \overline{n} &=_\beta (\lambda n\, f\, x.f((n\, f)\, x))(\lambda f\, x.f^n\, x) \\
&=_\beta \lambda f\, x.f(((\lambda g\, y.g^n y\, f)\, x \\
&=_\beta \lambda f\, x.f\, (\lambda y.f^n\, y)\, x \\
&=_\beta \lambda f\, x.f\, (f^n\, x) \\
&=_\beta \lambda f\, x.f^{n+1} x \\
&=_\beta \overline{n+1}
\end{aligned}$$

□

4.4.2 Les booléens

On définit les booléens et les conditions :

Notation 11.

$$\begin{aligned}
\text{true} &= \lambda x\, y.x \\
\text{false} &= \lambda x\, y.y \\
\text{ifThenElse} &= \lambda a\, b\, c.a\, b\, c
\end{aligned}$$

Proposition 21.

Pour tout λ-terme B et C

$$\begin{aligned}
\text{ifThenElse true } B\, C &=_\beta B \\
\text{ifThenElse false } B\, C &=_\beta C
\end{aligned}$$

Démonstration. On a, pour tout λ-terme B et C,

$$\begin{aligned}
\text{ifThenElse true } B\, C &=_\beta (\lambda a\, b\, c.a\, b\, c)\, (\lambda x\, y.x)\, B\, C \\
&=_\beta (\lambda x\, y.x)\, B\, C \\
&=_\beta B
\end{aligned}$$

et pareillement pour

$$\text{ifThenElse false } B\, C =_\beta C$$

□

4.4.3 Les opérations arithmétiques

Puis les fonctions arithmétiques de base

Notation 12.

$$\begin{aligned}
\text{isZero} &= \lambda n.n\,(\lambda x.\,\text{false})\ \text{frue}\\
\text{plus} &= \lambda m\,n.m\ \text{succ}\ n\\
\text{mult} &= \lambda m\,n\,f.m\,(n\,f)\\
\text{power} &= \lambda m\,n.n\,(\text{mult}\ m)\,1
\end{aligned}$$

Proposition 22.

$$\begin{aligned}
\text{isZero}\ \overline{0} &=_\beta \text{true}\\
\text{isZero}\ \overline{n+1} &=_\beta \text{false}\\
\text{plus}\ \overline{n}\ \overline{m} &=_\beta \overline{n+m}\\
\text{mult}\ \overline{n}\ \overline{m} &=_\beta \overline{nm}\\
\text{power}\ \overline{n}\ \overline{m} &=_\beta \overline{n^m}
\end{aligned}$$

Démonstration. On peut justifier ces notations en prouvant leur cohérence simplement par récurrence, par exemple. □

4.4.4 Les couples

Maintenant, nous allons nous traiter les structures mémoire. Tout d'abord, très classiquement, les couples.

Notation 13.

$$\begin{aligned} \text{pair} &= \lambda a\, b\, p.p\, a\, b \\ \text{first} &= \lambda a.a\ \text{true} \\ \text{second} &= \lambda a.a\ \text{false} \end{aligned}$$

Notation 14.

Étant donné t et u des λ-termes, on note

$$(u, v) = \text{pair}\ u\, v$$

On peut vérifier que

Proposition 23.

$$\begin{aligned} \text{first}\,(a, b) &=_\beta a \\ \text{second}\,(a, b) &=_\beta b \end{aligned}$$

Démonstration. Immédiat en développant first et second et en substituant. □

On peut alors définir la fonction prédécesseur

Notation 15.

$$\text{pred} := \lambda n.\,\text{first}\,(n\,(\lambda c.(\text{second}\ c, \text{succ}\,(\text{second}\ c)))(0, 0))\,.$$

Proposition 24.

$$\begin{aligned} \text{pred}\ \overline{0} &=_\beta \overline{0} \\ \text{pred}\ \overline{n+1} &=_\beta \overline{n} \end{aligned}$$

Démonstration. Le but du prédécesseur est de gardé un couple de la forme $(i - 1, i)$. Quand le second terme atteint n, le premier vaut $n - 1$. □

Notation 16.

$$\text{sub} = \lambda n\, p.p\ \text{pred}\ n$$

Proposition 25.

$$\text{sub}\ \overline{n}\,\overline{m} = \begin{cases} \overline{n-m} & n \geqslant m \\ \overline{0} & n < m \end{cases}$$

4.4.5 Les listes

On stockes les listes telles que $[a_1, \ldots, a_n] = \lambda g\, y.g\, a_1\, (\ldots (g\, a_n\, y) \ldots)$. Cela correspond à l'opération *fold right* dans les langages fonctionnels.

Notation 17.

$$\begin{aligned} \text{nil} &= \lambda f\, x.x \text{ aussi noté } [] \\ \text{cons} &= \lambda a\, l\, f\, x.f\, a\, (l\, f\, x) \\ A :: B &= \text{cons}\ A\, B \\ \text{isEmpty} &= \lambda a.a\, (\lambda x\, y.\, \text{false})\ \text{true} \\ \text{head} &= \lambda l.l\, (\lambda a\, b.a) \\ \text{tail} &= \lambda l.\, \text{first}\, (l\, (\lambda A\, B.(\text{second}\ B, A :: \text{second}\ B))\, ([], [])) \end{aligned}$$

Proposition 26.

$$\begin{aligned} \text{isEmpty}\ [] &=_\beta \text{true} \\ \text{isEmpty}\ A :: B &=_\beta \text{false} \\ \text{head}\ A :: B &=_\beta A \\ \text{tail}\ A :: B &=_\beta B \end{aligned}$$

4.4.6 Les arbres binaires

On définit une autre structure de mémoire : les arbres binaires. Le but est de les stocker sous une forme proche des listes.

Notation 18.

$$\begin{aligned}\text{feuille} &= \lambda n\, g\, y.y\, n\\ \text{noeud} &= \lambda b\, c\, g\, y.g\, (b\, g\, y)\, (c\, g\, y)\end{aligned}$$

Proposition 27.

Pour un arbre t et des λ-termes g et y, on a

$$t\, g\, y = \begin{cases} y\, n & \text{si } t \text{ est la feuille } n \\ g\, l\, g\, y\, r\, g\, y & \text{si } t \text{ a } l \text{ et } r \text{ comme fils} \end{cases}$$

4.4.7 La récursion

On s'intéresse maintenant à la constructions d'algorithmes. Les structures présentées jusqu'à présents sont très statiques : ce sont les valeurs et leurs itérateurs. On veut pouvoir faire des programmes plus complexes.

Le récurseur permet déjà des programmes plus complexes. C'est une fonction qui prend 3 arguments n, f et x et renvoie le $n^{\text{ème}}$ terme de la suite définie par

$$u_n = \begin{cases} x & n = 0 \\ f(n, u_{n-1}) & n > 0 \end{cases}$$

Le récurseur peut s'exprimer ainsi,

Notation 19.

$$\text{rec} = \lambda n\, f\, x.\, \text{first}\, (n\, (\lambda c.(f\, (\text{second}\ c)\, (\text{first}\ c), \text{succ}\, (\text{second}\ c)))\, (x, 0))$$

4.4.8 Le combinateur de point fixe

Un combinateur de point fixe sert à trouver un point fixe à une fonction donnée en argument.

Le plus simple, défini ainsi

Notation 20.

$$Y = \lambda f.(\lambda x.f\ (x\ x))(\lambda x.f\ (x\ x))$$

Ce célèbre combinateur est connu sous le nom de combinateur de point fixe de CURRY.

Proposition 28.

Pour tout λ-terme g

$$Y\ g =_\beta g\ (Y\ g)$$

Démonstration.

$$\begin{aligned} Y\ g &=_\beta (\lambda f.((\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x)))\ g \\ &=_\beta (\lambda x.g\ (x\ x))(\lambda x.g\ (x\ x)) \\ &=_\beta g\ ((\lambda x.g\ (x\ x))(\lambda x.g\ (x\ x))) \\ &=_\beta g\ (Y\ g) \end{aligned}$$

□

Grâce à ça on peut aussi définir le pgcd :

Notation 21.

$$\begin{aligned} \text{pgcd} := (\lambda f\ n\ p.\ &\text{ifThenElse}\ (\text{isZero}\ (\text{sub}\ \ p\ n)) \\ &(\text{ifThenElse}\ (\text{isZero}\ (\text{sub}\ \ n\ p)) \\ &\quad p \\ &\quad (f\ p(\text{sub}\ \ n\ p))) \\ &(f\ n\ (\text{sub}\ \ p\ n))) \end{aligned}$$

Proposition 29.

$$\text{pgcd}\ \overline{n}\,\overline{m} = \overline{n \wedge m}$$

Démonstration. En effet, on a

$$m \wedge n = \begin{cases} n & m = n \\ m \wedge (n - m) & n > m \\ (m - n) \wedge n & n < m \end{cases}$$

□

Chapitre 5

Les machines à accès arbitraire

Les machines à accès arbitraire sont un modèle de calcul dont le principal intérêt est de ressembler aux microprocesseurs. Les machines à accès arbitraire sont aussi appelées machine RAM (pour *Random-Access Machine*). Il s'agit d'un modèle automatique utilisant un programme au sens usuel du terme, avec une mémoire ressemblant à celle d'un ordinateur : des cases consécutives contenant chacune un entier. La grosse différence avec les machines de TURING est que la machine peut décider d'accéder à une case précise de sa mémoire immédiatement, d'où le nom du modèle. Cette fonctionnalité existe aussi sur les ordinateurs réels et est assurée par les pointeurs. Ce sont des entiers décrivant une adresse mémoire, à laquelle on peut donc accéder immédiatement. Une autre différence, quoique mineure en réalité est que chaque case peut contenir un entier (non borné) et non un simplement un symbole d'un alphabet. Cette liberté peut sembler plus puissante qu'elle ne l'est en réalité.

5.1 Définition

Une machine RAM est constituée
— d'un programme lui-même constitué d'une suite finie d'instructions numérotées à partir de zéro
— d'un registre spécial appelé compteur ordinal (*Program counter* en anglais) qui contient le numéro de la prochaine instruction à exécuter
— d'une suite infinie de registres numérotés à partir de zéro. Le registre de numéro n est noté R_n
— d'un registre spécial appelé accumulateur et noté A
— d'une bande d'entrée (flux d'entrée) sur laquelle la machine lit ses données
— d'une bande de sortie (flux de sortie) sur laquelle la machine écrit ses

résultats.

5.1.1 Les registres et l'accumulateur

Les registres et l'accumulateur constituent la mémoire de la machine. Chacune de ces mémoires contient un entier naturel. Avant l'exécution du programme tous les registres et l'accumulateur sont initialisés à la valeur 0.

5.1.2 Les instructions

Les instructions des machines RAM se divisent en quatre catégories :
— Manipulations de registres
— Opérations arithmétiques
— Ruptures de séquences
— Instructions d'entrées/sorties

Manipulation de registres

Il y a deux instructions de manipulation de registres : `load` et `store`. L'instruction `load` admet 3 syntaxes.
— `load # n` : l'accumulateur est fixé à `n`. C'est l'*adressage absolu*.
— `load n` : l'accumulateur prend la valeur contenue dans le registre R_n. C'est l'*adressage direct*.
— `load(n)` : l'accumulateur prend la valeur contenue dans le registre R_p où p est la valeur contenue dans le registre R_n. C'est l'*adressage indirect* aussi nommé *indirection*.

L'instruction `store` admet 2 syntaxes.
— `store n` : l'accumulateur est copié dans le registre R_n. C'est l'*adressage direct*.
— `store(n)` : l'accumulateur est enregistré dans le registre R_p où p est le contenu du registre R_n. C'est l'*adressage indirect*.

Opérations arithmétiques

Les machines RAM ne disposent que de 2 opérations arithmétiques : l'incrémentation et la décrémentation. Elles sont respectivement notées `incr` et `decr`. Ces instructions ne prennent pas d'argument.
— `incr` augmente de 1 la valeur de l'accumulateur.
— `decr` diminue de 1 la valeur de l'accumulateur si celui-ci est strictement positif. Si $A = 0$, `decr` n'a aucun effet.

On peut aussi doter les machines RAM d'autres opérations arithmétiques usuelles comme l'addition, la soustraction ou même la multiplication. La puissance de calcul n'en est bien sûr pas affectée puisqu'on peut construire un programme d'addition ou de multiplication. On a décidé de restreindre autant que possible le nombre d'instructions.

Ruptures de séquences

Les machines RAM ne possèdent que trois instructions de rupture de séquences : `jump`, `jz` et `stop`. Les deux premières instructions prennent un argument qui est un numéro d'instruction du programme. L'instruction `stop` ne prend pas d'argument. L'instruction `jump` est appelée saut inconditionnel. Cette instruction fixe le compteur ordinal à l'entier donné en paramètre. L'instruction `jz` est appelée saut conditionnel. Cette instruction fixe le compteur ordinal à la valeur donnée en paramètre si $A = 0$, sinon le compteur ordinal est simplement incrémenté. L'instruction `stop` arrête la machine, ce qui équivalent à fixer le compteur ordinal à l'entier suivant le numéro de la dernière instruction.

Instructions d'entrées/sorties

Les machines RAM ne possèdent que deux instructions d'entrées/sorties : `read` et `write`. Ces deux instructions ne prennent pas d'argument. L'instruction `read` provoque la lecture d'un entier sur la bande d'entrée et le résultat est placé dans l'accumulateur. Chaque lecture avance la tête de lecture de manière à ce que des instructions `read` successives lisent séquentiellement les données sur la bande. L'instruction `write` provoque l'écriture du contenu de l'accumulateur sur la bande de sortie.

5.1.3 Différences avec les microprocesseurs

La différence la plus évidente, qui est la différence commune entre tous les modèles de calcul et un programme le simulant, est le caractère fini de la mémoire des microprocesseurs. Un microprocesseur a, en outre, plus de types de registres et tous les calculs ne sont pas réalisés et stockés dans une unique mémoire.

L'autre différence, moins fondamentale, est que les microprocesseurs sont dotés de bien plus d'instructions et d'opérations arithmétiques. En pratique un processeur supporte pas moins de 50 instructions.

5.2 Exemples

Addition On construit tout d'abord un programme d'addition :

```
0    read       // Lecture du premier entier x
1    store 0    // Sauvegarde de x dans R0
2    read       // Lecture du second entier y
3    store 1    // Sauvegarde de y dans R1
4    load 0     // Début de la boucle de calcul
5    jz 12      // Test de sortie de boucle
6    decr       // À chaque itération de la boucle,
7    store 0    // R0 est décrémenté et R1 est incrémenté.
8    load 1     // La boucle s'arrête quand R0 contient 0.
9    incr       // R1 contient alors la somme de x et y.
10   store 1
11   jump 4     // Fin de la boucle de calcul
12   load 1
13   write      // Affichage de la somme
14   stop
```

Ainsi on peut désormais noter $R_p \leftarrow R_m + R_n$ le programme qui enregistre $R_m + R_n$ dans R_p. Ce qui nécessite des registre R_i et R_j libres. C'est bien sûr toujours possible car il y a une infinité de registres et qu'au bout d'un nombre fini d'instructions, seul un nombre fini de registre peuvent être occupés. On obtient alors le programme suivant :

```
0    load m
1    store i
2    load n
3    store j
4    load i
5    jz 12
6    decr
7    store i
8    load j
9    incr
10   store j
11   jump 4
12   load j
13   store p
14   load# 0
15   store i
16   store j
```

Multiplication On peut par conséquent et de façon plus simple écrire le programme $R_p \leftarrow R_m \times R_n$:

```
0    load m
1    store i
2    load n
3    store j
4    load i
5    jz 10
6    decr
7    Rj <- Rj + Rn
8    jump 4
9    load j
10   store p
11   load# 0
12   store i
13   store j
```

5.3 Simulation en Python

Pour la simulation, on choisit d'écrire le programme sans spécifier les numéros d'instruction. On permet aussi les commentaires commençant par "//". On définit tout d'abord les types codant un registre et une machine RAM.

```
class CRegistre:
    def __init__(self):# Constructeur
        self._reg=0

    def incr(self):# Pour simuler l'incrementation
        self._reg+=1

    def decr(self):# Pour simuler la decrementation
        self._reg-=1

    def isZero(self):# Pour les sauts conditionnel
        return (self._reg==0)

    def setReg(self, i):
        self._reg=i

    def getReg(self):
        return self._reg

    reg=property(getReg, setReg)# Accesseur et mutateur

class RAM:
    """
    Registre
    Accumulateur
    Compteur ordinal
    """

    def __init__(self):# Constructeur
        self.registre=list()
        self.accumulateur=CRegistre()
        self.co=0

    def exe(self, prog, entree, sortie):# Fonction de simulation
        n=len(prog)
        """
        On arrete la simulation lorsque le compteur ordinal depasse la taille du programme.
        Pour cela on fera en sorte que l'instruction stop fixe le compteur ordinal juste apres
        la derniere instruction
        """
        while self.co<n:
            interpreter(self, prog[self.co], entree, sortie, n)
```

On définit maintenant la fonction d'interprétation des instructions permettant de faire fonctionner la machine.

```
43  def resize(machine, i):# Permet d'avoir une memoire d'une taille toujours suffisante
44      while len(machine.registre)<=i:
45          machine.registre.append(CRegistre())
46
47  def interpreter(machine, instr, entree, sortie, n):# Interpretation d'une instruction
48      """
49      Les instructions sont prealablement traitees et presentees dans un tableau de tableaux.
50      Chaque paire correspond a une instruction. La premiere composante contient l'instruction
51      et la seconde contient l'argument eventuel. Si aucun argument n'est requis, cet element
52      vaut 0. Les instructions sont ecrites de facon a differencier les differentes syntaxe
53      d'une meme instruction.
54      """
55
56      if instr[0]=="load#":
57          machine.accumulateur.setReg(instr[1])
58      elif instr[0]=="load":
59          if len(machine.registre)<=instr[1]:
60              resize(machine, instr[1])
61
62          machine.accumulateur.setReg(machine.registre[instr[1]].getReg())
63      elif instr[0]=="load(":
64          if len(machine.registre)<=instr[1]:
65              resize(machine, instr[1])
66          if len(machine.registre)<=machine.registre[instr[1]]:
67              resize(machine, machine.registre[instr[1]])
68
69          machine.accumulateur.setReg(machine.registre[machine.registre[instr[1]]].getReg())
70      elif instr[0]=="store":
71          if len(machine.registre)<=instr[1]:
72              resize(machine, instr[1])
73
74          machine.registre[instr[1]].setReg(machine.accumulateur.getReg())
75      elif instr[0]=="store(":
76          if len(machine.registre)<=instr[1]:
77              resize(machine, instr[1])
78          if len(machine.registre)<=machine.registre[instr[1]]:
79              resize(machine, machine.registre[instr[1]])
80
81          machine.registre[machine.registre[instr[1]]].setReg(machine.accumulateur.getReg())
82      elif instr[0]=="incr":
83          machine.accumulateur.incr()
84      elif instr[0]=="decr":
85          machine.accumulateur.decr()
86      elif instr[0]=="jump":
87          machine.co=instr[1]-1
88      elif instr[0]=="jz":
89          if machine.accumulateur.isZero():
90              machine.co=instr[1]-1
91
92      elif instr[0]=="stop":
93          machine.co=n
94      elif instr[0]=="read":
95          machine.accumulateur.setReg(entree[0])
96          del entree[0]
97      elif instr[0]=="write":
98          sortie.append(machine.accumulateur.getReg())
```

```
99
100     machine.co=machine.co+1
```

On donne les fonctions permettant d'analyser un texte brut pour renvoyer un programme et un ruban.

```
94  def analyse_prog(raw):
95      """
96      C'est la fonction qui analyse un texte brut codant un programme
97      pour en faire le tableau deja evoque.
98      """
99      prog=list()
100     i=0
101     while i<len(raw):
102         first=i
103         while i<len(raw) and raw[i]>='a' and raw[i]<='z':
104             i+=1
105 
106         second=i
107         w=0
108         qualificatif=""
109         if i<len(raw) and raw[i]==' ':
110             if raw[i+1]=='(':
111                 qualificatif="("
112             elif raw[i+1]=='#':
113                 qualificatif="#"
114 
115         while i<len(raw) and (raw[i]<'0' or raw[i]>'9') and raw[i]!='\n' and raw[i]!='/':
116             i+=1
117 
118         while i<len(raw) and raw[i]>='0' and raw[i]<='9':
119             w=w*10+ord(raw[i])-48
120             i+=1
121         prog.append([raw[first:second]+qualificatif,w])
122         while i<len(raw) and raw[i]!='\n':
123             i+=1
124         i+=1
125     return prog
126 
127 def analyse_entree(raw):
128     """
129     Cette fonction analyse le contenu du fichier servant de ruban d'entree et
130     renvoie un tableau d'entiers.
131     """
132     flux=list()
133     i=0
134     while i<len(raw):
135         w=0
136         while i<len(raw) and raw[i]>='0' and raw[i]<='9':
137             w=w*10+ord(raw[i])-48
138             i+=1
139         flux.append(w)
140         while i<len(raw) and raw[i]==' ':
141             i+=1
142     return flux
```

On donne le code gérant l'interface.

```
file_name=raw_input("Programme a executer : ")# Lecture du programme
file=open(file_name+".ram", "r")
prog=file.read()
file.close()

file_name=raw_input("Ruban d'entree :")# Lecture du ruban d'entree
file=open(file_name+".rbn", "r")
entree=file.read()
file.close()

flux_entree=analyse_entree(entree)# Analyse du texte brut pour en faire un tableau

m=RAM()

flux_sortie=[]

print("Ruban en entree :")
print(flux_entree)

m.exe(analyse_prog(prog), flux_entree, flux_sortie)# Execution de la machine

print("Ruban en sortie :")
print(flux_sortie)# Affichage du ruban de sortie
```

On peut donner comme programme (`add.ram`) :

```
read // Lecture du premier entier x
store 0 // Sauvegarde de x dans R0
read // Lecture du second entier y
store 1 // Sauvegarde de y dans R1
load 0 // Debut de la boucle de calcul
jz 12 // Test de sortie de boucle
decr // A chaque iteration de la boucle,
store 0 // R0 est decremente et R1 est incremente.
load 1 // La boucle s'arrete quand R0 contient 0.
incr // R1 contient alors la somme de x et y.
store 1
jump 4 // Fin de la boucle de calcul
load 1
write // Affichage de la somme
stop
```

Et comme ruban d'entrée (`in.rbn`) :

```
42 1337
```

On obtient ainsi dans la console d'exécution :

```
Programme a executer : add
Ruban d'entree :in
Ruban en entree :
[42, 1337]
Ruban en sortie :
[1379]
```

Chapitre 6

Les machines à compteurs

6.1 Définition

Les machines à compteurs constituent le modèle de calcul le plus simple. On les appelle aussi machines à registres ou machines de MINSKY.

Une machine à compteurs est constituée de n compteurs (aussi appelés registres) et d'un programme.

C_i désigne le $i^{\text{ème}}$ compteur et i_1 désigne la $i^{\text{ème}}$ instruction.

Le programme est constitué de 3 types d'instructions :

— Incrémente C_i puis passe à l'instruction suivante

— Décrémente C_i puis passe à l'instruction suivante

— Si $C_1 = 0$ alors saut vers l'instruction i_1 sinon saut vers l'instruction i_2

On précise que la décrémentation de 0 donne 0.

6.2 Simulation en OCaml

La simplicité de la simulation est à l'image de la simplicité du modèle de calcul.

`Définition des types`

```
type instruction=Inc of int | Dec of int | Saut of int*int*int;;
(* On definit les 3 types d'instructions *)
(* Pour l'incrementation et la decrementation on donne seulement le numero du compteur *)
(* Pour Saut(i,j,k) i represente le compteur a tester, j l'instruction a laquelle sauter
si le compteur i est nul et k le numero de l'instruction ou aller dans le cas contraire *)

type machine={programme : instruction array;
              nbr : int};;
(* La description d'une machine contient les instructions et le nombre de compteurs *)

exception Hors_Champ1 of int array*int*int;;
exception Hors_Champ2 of int array*int*int;;
exception Non_defini of int array*int*int;;

(* On gere egalement les cas ou on appellerai une instruction ou un compteur inexistant *)
```

Fonctions d'exécutions

```
let execution machine=
    let comp=(Array.create machine.nbr 0) in
    let i=ref 0 in
    while !i<Array.length machine.programme do
         match machine.programme.(!i) with
         |Inc a->   if a<0 || a>=machine.nbr then
                        raise (Non_defini(comp,a,!i));
                    comp.(a)<-comp.(a)+1;
                    incr i
         |Dec a->   if a<0 || a>=machine.nbr then
                        raise (Non_defini(comp,a,!i));
                    comp.(a)<-max (comp.(a)-1) 0;
                    incr i
         |Saut (a,b,c)->if b<0 || b>=Array.length machine.programme then
                            raise (Hors_Champ1(comp,b,!i))
                        else if c<0 || c>=Array.length machine.programme then
                            raise (Hors_Champ2(comp,c,!i));
                        if comp.(a)=0 then
                            i:=b
                        else
                            i:=c
    done;
    comp;;

let exe machine=
    let comp=ref [||] in
    try
    (
        comp:=execution machine;
        for i=0 to (Array.length !comp)-1 do
            print_string "Compteur ";
            print_int i;
            print_string " : ";
            print_int !comp.(i);
            print_newline()
        done;
    )(* Execution suivi de l'affichage du resultat *)
    with
        |Hors_Champ1 (t,a,b)->  (print_string "Erreur : l.";
                                 print_int b;
                                 print_newline();
                                 print_string "2eme argument : La ligne ";
                                 print_int a;
                                 print_string " n'existe pas.";
                                 print_newline();)
        |Hors_Champ2 (t,a,b)->  (print_string "Erreur : l.";
                                 print_int b;
                                 print_newline();
                                 print_string "3eme argument : La ligne ";
                                 print_int a;
                                 print_string " n'existe pas.";
                                 print_newline();)
        |Non_defini (t,a,b)->  (print_string "Erreur : l.";
                                 print_int b;
                                 print_newline();
                                 print_string "1eme argument : Le compteur ";
                                 print_int a;
```

```
58                                    print_string " n'existe pas.";
59                                    print_newline();;;
60  (* Gestion des exceptions sous la forme de messages d'erreurs *)
```

Chapitre 7

Les automates à piles multiples

7.1 Définition

On va s'intéresser dans ce chapitre aux automates à n piles. C'est une extension des automates à une pile, modèle de calcul bien connu en théorie des automates. Ce modèle est assez peu répandu et ne présente que peu d'intérêt. On s'en servira ici comme intermédiaire entre différents modèles de calcul afin de prouver leur TURING-équivalence.

Définition 38.

Un automate à n piles est un 7-uplet $(Q, \Sigma, \mathcal{Z}, q_0, z_0, \delta F,)$ où :
- Q est un ensemble fini d'états
- Σ, un alphabet fini, appelé l'alphabet d'entrée
- $\mathcal{Z}$, un alphabet fini, a priori distinct de Σ, appelé l'alphabet de pile
- $q_0 \in Q$, l'état initial
- $z_0 \in \mathcal{Z}$, le symbole de fond de pile
- $F \subseteq Q$, les états acceptants
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \mathcal{Z}^n \to Q \times \mathcal{Z}^n$, la fonction de transition.

Pour être plus précis, δ est une fonction partielle. De plus, pour des raisons que l'on verra par la suite, on impose 2 contraintes. Lorsque $\delta(q, \varepsilon, z)$ est définie, $\delta(q, a, z)$ n'est définie pour aucune lettre $a \in \Sigma$. En effet, s'il les deux transitions sont simultanément définies, elles conviennent toutes deux pour au moins une lettre de Σ donc pour au moins un mot de $\Sigma^\star$. On perd ainsi le déterminisme de l'automate.

D'autre part, on impose une contrainte sur la définition de la fonction portant sur la lecture des piles : lorsque $\delta(q, a, (z_1, \ldots, z_0, \ldots, z_n))$ est définie, δ n'est définie en aucun point de la forme $(q, a, (z_1, \ldots, z_i, \ldots, z_n))$ pour $z_i \in \mathcal{Z}$, et ce quel que soit le nombre de z_0 intervenants dans le n-uplet. Cette contrainte aussi permet de respecter le déterminisme de l'automate.

Ainsi et plus généralement, c'est uniquement l'état q qui détermine s'il faut lire une lettre et quelles sont les piles à prendre en compte.

D'autre part, la pertinence de cette définition provient du fait que lorsque $n = 1$, on a obtient la définition des automates à pile unique que l'on verra plus loin.

7.2 Mode de fonctionnement

Un automate à n piles a une mémoire organisée en n piles LIFO distincts d'éléments de $\mathcal{Z}$. Une pile LIFO est une suite finie d'éléments de $\mathcal{Z}$ qui peut prendre des tailles arbitrairement grandes et dont seul l'élément du dessus est accessible à la lecture. L'automate travaille sur un mot de $\Sigma^\star$ qui est lu lettre par lettre.

À l'état initial, l'automate se trouve dans l'état q_0. Ensuite l'automate effectue des transitions. On peut distinguer plusieurs types de transitions. On considère la transition $(q, a, (z_1, \ldots, z_n)) \mapsto (q', (y_1, \ldots, y_n))$. On distingue d'abord 2 cas selon la valeur de a. Si $a = \varepsilon$, le mot n'est pas lu et à l'issue de la transition, le mot n'a pas changé. On parle de ε-règle. Si $a \neq \varepsilon$, la première lettre du mot est prise en compte et alors le mot à l'issue de la transition ne contient plus la première lettre. Il apparait la justification de la première contrainte.

On peut faire une autre distinction selon la valeur des symboles lus ou écrits sur les piles. Si $z_i = z_0$, le symbole au sommet de la pile n'est pas lu et on ne dépile rien. Si, au contraire, le symbole lu n'est pas le symbole de fond de pile, l'élément est dépilé à sa lecture. Ce qui explique la seconde contrainte.

7.3 Simulation en C

On propose un programme écrit en C permettant de simuler les machines à plusieurs piles. Les piles peuvent être représentées par des tableaux au des listes chainées, mais la simplicité de l'emploi des tableau en a fait le meilleur candidat. Il sera donc nécessaire de recourir à l'allocation dynamique.

On commence par définir la structure codant les transitions et les machines :

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct transition transition;
struct transition
{
    char lettre;
    int* sommets;
    int etat;
    int etat_resultant;
    int* sommets_resultants;
};

typedef struct multipile multipile;
struct multipile
{
    int n;
    int trans_nb;
    transition* table;
    int nb_acceptant;
    int* acceptant;
    int etat_init;
};
```

On utilisera des `int` comme lettres de l'alphabet de travail et pour les états et des `char` pour l'alphabet du mot. On considère en outre que l'alphabet de travail ne contient pas 0. Une comparaison à 0 sur le sommet d'une pile correspond à l'absence de test et donc au maintient du sommet sans dépilage ; il en va de même pour l'empilage d'une valeur : un 0 dans un sommet résultant d'une transition correspond à l'absence d'empilage.

On donne la fonction qui, sachant la configuration d'une machine, trouve l'index de la transition correspondante. Cette fonction revoie -1 si la transition n'existe pas. La machine s'arrête alors sur un calcul non réussi.

```
int find_trans(int n, transition* table, int trans_nb, int** piles,
        int* tailles, char lettre, int etat)
{
    int test=0;
    int conv=-1;
    int i=0;
    int j=0;

    for(i=0;i<trans_nb;++i)
    {
        if(lettre==table[i].lettre && etat==table[i].etat)
        {
            test=0;
            for(j=0;j<n;++j)
            {
                if(table[i].sommets[j]!=0 && tailles[j]==0)
                    test=1;
                if(tailles[j]>0)
                    if(table[i].sommets[j]!=0 &&
                            table[i].sommets[j]!=piles[j][tailles[j]-1])
                        test=1;

            }
            if(test==0)
            {
                conv=i;
                break;
            }
        }
    }

    if(conv==-1)
        return -1;

    for(i=0;i<n;++i)
        if(table[conv].sommets[i]!=0 && tailles[i]>0)
        {
            tailles[i]--;
            piles[i]=(int*)realloc(piles[i],tailles[i]*sizeof(int));
        }
    return conv;
}
```

On donne maintenant la fonction d'exécution qui renvoie un entier : 1 si le mot est reconnu, -1 si le calcul provoque une erreur (absence de transition), 0 si le mot n'est pas reconnu.

```
66  int exec(multipile machine, int mot_size, char* mot)
67  {
68      int** piles=NULL;
69      int* tailles=NULL;
70      int progression=0;
71      int i=0;
72      int l=0;
73      int etat=machine.etat_init;
74
75      piles=(int**)calloc(machine.n,sizeof(int));
76      tailles=(int*)calloc(machine.n,sizeof(int));
77
78      for(l=0;l<machine.n;++l)
79          tailles[0]=0;
80
81      while(progression<mot_size)
82      {
83          i=find_trans(machine.n, machine.table, machine.trans_nb, piles,
84                  tailles, mot[progression], etat);
85          if(i==-1)
86              return -1;
87          etat=machine.table[i].etat_resultant;
88          for(l=0;l<machine.n;++l)
89          {
90              if(machine.table[i].sommets_resultants[l]!=0)
91              {
92
93                  tailles[l]++;
94                  piles[l]=(int*)realloc(piles[l],tailles[l]*sizeof(int));
95                  piles[l][tailles[l]-1]=machine.table[i].sommets_resultants[l];
96              }
97          }
98          ++progression;
99      }
100
101     for(l=0;l<machine.nb_acceptant;++l)
102         if(machine.acceptant[l]==etat)
103             return 1;
104
105     return 0;
106 }
```

On donne aussi un exemple simple reconnaissant les mots sur l'alphabet $\{a, b\}$ ne contenant pas de b. De plus la première pile contient au final une case par a lu et la seconde pile une case par b lu.

```
107 int main(void)
108 {
109     int a[]={0,0};
110     int b[]={0,1};
111     int c[]={1,0};
112     int term[]={0};
113 
114     transition tr_1={'a',a,0,0,b};
115     transition tr_2={'a',a,1,1,b};
116     transition tr_3={'b',a,0,1,c};
117     transition tr_4={'b',a,1,1,c};
118     transition table[]={tr_1,tr_2,tr_3,tr_4};
119 
120     multipile machine={2,4,table,1,term,0};
121 
122     printf("%d\n",exec(machine,4,"aaaa")); //Renvoie 1
123     printf("%d\n",exec(machine,4,"aaba")); //Renvoie 0
124     printf("%d\n",exec(machine,4,"bbbb")); //Renvoie 0
125 
126     system("PAUSE");
127 
128     return 0;
129 }
```

Deuxième partie

Équivalence des modèles de calcul

Chapitre 8

TURING-équivalence des automates cellulaires

8.1 Simulation d'une machine de TURING à l'aide d'un automate cellulaire

Proposition 30.

Pour toute machine de TURING, il existe un automate cellulaire qui réalise le même calcul.

Démonstration. Soit $M = (Q, \Gamma, B, \Sigma, q_0, \delta, F)$ une machine de TURING.

On va construire un automate cellulaire à une dimension qui réalise le même calcul. Le contenu de chaque case représentera le contenu du ruban.

L'ensemble des état de l'automate, noté $\mathcal{Q}$ est $\Gamma \cup (\Gamma \times Q)$. Le voisinage v est $\{-1, 0, 1\}$.

La règle de l'automate est telle qu'elle simule une transition de la machine de TURING. Il n'y aura qu'une case contenant un élément de $\Gamma \times Q$ et elle représentera la tête de lecture. Toutes les autres cases contiennent un élément de Γ.

Aussi la règle locale respecte les conditions suivantes où les a désignes des éléments de Γ et q des éléments de Q.

- $(a_G, a, a_D) \to a$
- $(a_G, (a, q), a_D) \to (a', q')$ si $\delta(q, a) = (q', a', m)$ et $m \neq \texttt{HALT} \wedge q \notin F$
- $(a_G, (a, q), a_D) \to (a, q)$ si $\delta(q, a) = (q', a', m)$ avec $m = \texttt{HALT} \vee q \in F$
- $(a_G, a, (a_D, q)) \to$
  - (a, q') si $\delta(q, a) = (q', a', \longleftarrow)$

    - a sinon.
- $((a_G, q), a, a_D) \to$
    - (a, q') si $\delta(q, a) = (q', a', \longrightarrow)$
    - a sinon.

On a ainsi la même évolution de l'automate et de la machine de TURING. De plus, si la machine s'arrête, l'automate se stabilise. Réciproquement, deux configuration successives identiques suffit à dire que l'automate s'est stabilisé et que la machine s'est arrêté. Ainsi, on a une équivalence entre la stabilisation de l'automate et l'arrêt de la machine de TURING.

□

Corollaire 3.

La stabilisation des automates cellulaires est indécidable.

Démonstration. Si elle était décidable, grâce à la réduction précédente, l'arrêt des machines de TURING le serait aussi. Or, ce n'est pas le cas. Donc la stabilisation est indécidable. □

Corollaire 4.

Les automates cellulaires sont TURING-complets.

Démonstration. Il suffit d'appliquer le résultat précédent à une machine de TURING universelle. On obtient ainsi un automate cellulaire TURING-universel. □

8.2 Supériorité des machines de TURING sur les automates cellulaires

On va maintenant justifier que les machines de TURING sont plus puissantes que les automates cellulaires. On ne va pas donner de machines explicites, les fonctions de transitions seraient bien trop fastidieuses à décrire, mais seulement des arguments permettant de construire une telle machine.

Proposition 31.

Pour tout automate cellulaire, il existe une machine de TURING qui le simule

Démonstration. Tout d'abord, il faut se convaincre qu'on peut appliquer la règle locale sur une cellule. Comme le voisinage est fini, il y a un nombre fini de possibilités : $|Q|^{|v|}$. Il suffit d'accéder à ces $|v|$ cellules et de choisir la transition correcte amenant vers le bon état.

Ensuite, il faut montrer qu'on peut appliquer la règle globale.

Pour un calcul, un automate cellulaire commence avec une configuration finie pour un certain état. Donc, au delà de la taille de la configuration finie à laquelle on ajoute deux fois le rayon du voisinage par génération, le comportement des cellules est le même. Aussi, il n'y a qu'une quantité finie d'information à mémoriser : la taille de configuration finie et l'état des cellules hors d'atteinte de cette configuration. Il suffit donc d'appliquer la règle finie un nombre fini de fois à chaque étape.

On peut donc simuler un tel automate avec une machine de TURING. Les automates cellulaires ne sont donc pas plus puissants que les machines de TURING. □

8.3 Conclusion

Théorème 5.

La classe des fonctions calculables par l'ensemble des automates cellulaires à configuration initiale finie est égale à la classe des fonctions calculables par les machines des TURING.

Démonstration. ($\Rightarrow$) : On peut construire une machine de TURING pour simuler un automate cellulaire.

($\Leftarrow$) : On a construit un automate cellulaire pour chaque machine de TURING, en particulier, un qui simule une machine de TURING universelle est qui est donc lui-même TURING-complet.

Aussi, ces deux classes de fonctions sont égales. □

Chapitre 9

TURING-équivalence des fonctions récursives

9.1 Construction des fonctions récursives à l'aide de machines de TURING

On veut montrer qu'il est possible de construire les fonctions de bases ainsi que les différentes méthodes de constructions des fonctions.

9.1.1 La fonction nulle

On peut construire une machine de TURING qui efface une séquence consécutive de C-cases codant un n-uplet puis qui écrit 0. Pour créer une telle machine on peut se référer aux m-fonctions déjà écrites.

9.1.2 Les projections

On va construire la projection $p_{k,i} = ((n_1, \ldots, n_k) \mapsto n_i)$. On débute dans l'état q_1 sur la case contenant n_1. Pour tout $l \in [\![1, i]\!]$, on réalise la transition $(q_l, quelconque) \mapsto (q_{l+1}, 0, \rightarrow)$. On réalise alors la transition $(q_i, quelconque) \mapsto (q_{i+1}, \#, \rightarrow)$, puis on reprend $(q_l, quelconque) \mapsto (q_{l+1}, 0, \rightarrow)$ en choisissant que q_{k+1} est acceptant.

9.1.3 Les fonctions de duplication

Pour la fonction de duplication, on itère la composée de la m-fonction *pe* (d'écriture à la fin) et du décalage à gauche. On copie ainsi un nombre donné de fois l'élément présent au début du ruban.

9.1.4 La fonction successeur

Ici on peut distinguer deux cas :
— Le nombre est un élément de l'alphabet et celui-ci est ordonné.
— Le nombre est décrit dans une décomposition donnée sur le ruban, par exemple en binaire.

1er cas Il suffit de faire une transition pour chaque cas. Cette modélisation a un défaut majeur : il n'y a qu'un nombre fini de nombres et elle demande de nouvelles machines au fur et à mesure des successions.

2nd cas On suppose la machine fonctionnant avec un demi-ruban. On utilise une architecture big-endian : les bits de poids faible sont à gauche du ruban. La fonction consiste à parcourir le ruban vers la droite en inversant tous les bits trouvés. Si on trouve un bit égal à 0, la machine s'arrête après cette inversion.

9.1.5 La composition

La composition de machine de TURING est une opération aisée. Il suffit de construire une troisième machine qui ferait pointer tous les états acceptants de la première machine sur l'état initial de la deuxième. En outre cette machine a comme alphabet la réunion des deux alphabets et a pour ensemble des états acceptants celui de la seconde machine. Ainsi à un mot du ruban, on applique successivement les deux machines, ce qui revient à composer les fonctions simulées par les deux machines.

9.1.6 La construction récursive

La construction récursive se fait à partir de la composition et des tests qu'on peut facilement réaliser avec les tables abrégées.

9.1.7 La minimisation

Pour la minimisation, il suffit de réaliser tous les tests dans l'ordre. Si la machine ne s'arrête pas, alors la minimisation n'était pas définie en ce point. En effet si la minimisation est définie en ce point, alors il existe un entier qui annule la fonction. Cet entier sera nécessairement trouvé en un nombre fini de transitions donc la machine s'arrêtera.

9.2 Simulation des machines de TURING à l'aide des fonctions récursives

Nous allons maintenant montrer qu'une fonction calculable par une machine de TURING est récursive. Soit M une machine de TURING. On suppose que la machine ne se bloque jamais. Soit n le nombre d'états de M et soit m le cardinal de son alphabet de bande. On suppose que les états de M sont les entiers $[\![0, n-1]\!]$ et que les symboles de bande sont les entiers $[\![0, m-1]\!]$ le symbole blanc # étant l'entier 0. Pour tout mot $w = a_0 \dots a_k$ sur l'alphabet de bande, on associe les deux entiers $\alpha(w) = \sum_{i=0}^{k} a_i m^{k-i}$ et $\beta(w) = \sum_{i=0}^{k} a_i m^i$ obtenus en lisant w comme un nombre écrit en base m (avec les unités respectivement à droite et à gauche pour α et β). Les fonctions α et β permettent de considérer les mots de la bande comme des entiers.

Soit w une entrée de M et soit $C_0 \to \dots \to C_n$ le calcul de M sur w. On suppose que chaque configuration C_k est égale à $u_k q_k v_k$ où u_k, q_k et v_k représentent respectivement le ruban à gauche strictement de la tête de lecture, l'état de la machine et le ruban à droite de la tête de lecture. On va montrer que la fonction c définie par :

$$c(\beta(w), k) = (\alpha(u_k), q_k, \beta(v_k))$$

est primitive récursive. À partir des transitions de M, on peut définir une fonction t telle que :
$t(q,a) = (p,b,x)$ où $(q,a) \to (p,b,x)$ est une transition de M.

On complète t sur $\mathbb{N}^2$ en posant $t(i,j) = (0,0,0)$ si $i \geqslant n$ ou $j \geqslant m$. La fonction t est bien sûr primitive récursive. On note t_1, t_2 et t_3 les fonctions $p_1(t)$, $p_2(t)$ et $p_3(t)$ qui donnent respectivement le nouvel état, le symbole à écrire et le déplacement de la tête de lecture. On a alors la définition récursive de c.

$$c(\beta(w), 0) = (0, q_0, \beta(w))$$

Soit $a_k = mod(\beta(v_k), m)$. Si $t_3(q_k, a_k) = \longrightarrow$
— $\alpha(u_{k+1}) = m\alpha(u_k) + t_2(q_k, a_k)$
— $q_{k+1} = t_1(q_k, a_k)$
— $\beta(v_{k+1}) = div(\beta(v_k), m)$

Soit $a_k = mod(\beta(v_k), m)$. Si $t_3(q_k, a_k) = \longleftarrow$
— $\alpha(u_{k+1}) = div(\alpha(u_k), m)$
— $q_{k+1} = t_1(q_k, a_k)$
— $\beta(u_{k+1}) = m(\beta(v_k) - a_k + t_2(q_k, a_k)) + mod(u_k, m)$

9.3 Conclusion

Théorème 6 (TURING-équivalence des fonctions récursives)**.**

Une fonction de $\mathbb{N}^k$ dans $\mathbb{N}^r$ est récursive si, et seulement si, elle est calculable au sens de TURING.

Démonstration. $(\Rightarrow)$: On a en effet prouvé que les fonctions calculables par une machines de TURING peuvent être décrites avec des fonctions récursives.

$(\Leftarrow)$: On a aussi montré que les fonctions de base et les modes de construction des fonctions récursives sont calculables au sens de TURING donc les fonctions récursives sont calculables au sens de TURING.

Les machines de TURING sont donc équivalentes aux fonctions récursives. □

Chapitre 10

TURING-équivalence du λ-calcul

On va utiliser une méthode de démonstration assez inhabituelle : utiliser des langages de programmation. On utilisera dans un premier temps le Haskell. Le Haskell est un langage fonctionnel pur assez peu répandu dont le principal intérêt est d'être très proche du λ-calcul. On admet dans un premier temps qu'il est tout à fait équivalent au λ-calcul. On montrera en annexe que les fonctions utilisées peuvent être créées en utilisant le λ-calcul. Le second langage dont on aura besoin est le Brainfuck. Ce langage est un langage minimaliste et exotique dont l'intérêt principal réside dans le fonctionnement qui est très proche d'une machine de TURING. On montrera l'équivalence en annexe. Il s'agit donc d'écrire en Haskell un programme capable d'interpréter le Brainfuck.

10.1 Description du Brainfuck

Le Brainfuck, étant un langage exotique, n'a jamais bénéficié d'une norme. On donnera ici une version très proche des machines de TURING. Le Brainfuck utilise une unique mémoire organisée en ruban et munie d'une tête de lecture/écriture. Chaque case du ruban contient un entier naturel, initialement nul. On dispose de 8 instructions chacune codée par un caractère.

Symbole	Instruction
+	Incrémente la valeur courante
-	Décrémente la valeur courante
<	Décalage de la tête de lecture à gauche
>	Décalage de la tête de lecture à droite
[	Saute au] correspondant si la valeur courante est non nulle
]	Saute au [correspondant
.	Affichage de la valeur courante via la sortie standard
,	Affectation de la valeur courante via l'entrée standard

L'instruction '.' peut avoir un intérêt pour écrire le résultat au fur et à mesure du calcul, par exemple pour afficher les décimales successives d'un nombre calculable.

En outre l'instruction ',' ne sert pas pour les machines de TURING déterministes, comme toutes celles décrites jusque là. Cet instruction à un intérêt pour les machines de TURING non déterministes, que l'on verra plus tard. Par conséquent l'implémentation de l'interpréteur en Haskell ne prendra pas en compte la dernière instruction, qui n'est pas nécessaire pour avoir la TURING-équivalence.

10.2 Implémentation de l'interpréteur

On veut faire une fonction de simulation qui, étant donné le ruban strictement à gauche de la tête de lecture, le ruban à droite et la liste des instructions, renvoie le triplet constitué du ruban à gauche strictement, à droite et la liste des valeurs dans la sortie standard.

On procède pour cela en 2 temps. D'abord on constitue la correspondance des crochets. On crée une liste contenant des pairs de valeurs ? lesquelles correspondent aux positions des crochets associés.

```
dernier_non_assoc corr pos=case corr of
                                t:q->if snd t == -1 then
                                        (fst t,pos):q
                                    else
                                        t:(dernier_non_assoc q pos)


parenthesage instruction corr pos=    case instruction of
                                        []->corr
                                        '[':q-> parenthesage q ((pos,-1):corr) (pos+1)
                                        ']':q-> parenthesage q (dernier_non_assoc corr pos) (pos+1)
                                        _:q-> parenthesage q corr (pos+1)
```

On définit alors les fonctions permettant de trouver la position associée à une position donnée en parcourant la liste préalablement construite.

```
assoc ((a,b):q) e --Normalement le cas de la liste vide n'arrive pas.
    |a==e = b
    |otherwise = assoc q e

reciproque ((a,b):q) e --Idem
    |b==e = a
    |otherwise = reciproque q e
```

On peut ainsi définir la fonction de simulation. On gère les cas nécessitant un agrandissement du ruban.

```
exec gauche droit instruction pos correspondance cout
  |length instruction<=pos = (gauche,droit,cout)
  |length instruction>pos = case (gauche,droit,instruction !! pos) of
    (_,td:qd,'+')->exec gauche ((td+1):qd) instruction (pos +1) correspondance cout
    (_,[],'+')->exec gauche [1] instruction (pos +1) correspondance cout
    (_,td:qd,'-')->exec gauche ((td-1):qd) instruction (pos +1) correspondance cout
    (_,[],'-')->exec gauche [-1] instruction (pos +1) correspondance cout
    (tg:qg,_,'<')->exec qg (tg:droit) instruction (pos +1) correspondance cout
    ([],_,'<')->exec [] (0:droit) instruction (pos +1) correspondance cout
    (_,td:qd,'>')->exec (td:gauche) qd instruction (pos +1) correspondance cout
    (_,[],'>')->exec (0:gauche) [] instruction (pos +1) correspondance cout
    (_,[],']')->exec gauche droit instruction (pos +1) correspondance cout
    (_,td:qd,']')->
        if td==0 then
            exec gauche droit instruction (pos +1) correspondance cout
        else
            exec gauche droit instruction (reciproque correspondance pos) correspondance cout
    (_,[],'[')->exec gauche droit instruction (assoc correspondance pos) correspondance cout
    (_,td:qd,'[')->
        if td==0 then
            exec gauche droit instruction (assoc correspondance pos) correspondance cout
        else
            exec gauche droit instruction (pos +1) correspondance cout
    (_,t:q,'.')->exec gauche droit instruction (pos +1) correspondance (t:cout)



sim gauche droit instruction = exec gauche droit instruction 0 (parenthesage instruction [] 0) []
```

On voit clairement que la fonction `sim` correspond bien à l'exécution du Brainfuck. On a ainsi un code transposable en λ-calcul qui simule une machine de TURING, donc les machines de TURING ne peuvent pas être plus puissantes que le λ-calcul.

10.3 Implémentation du λ-calcul dans un système TURING-équivalent

Le but est d'implémenter un système de simulation du λ-calcul grâce à un modèle TURING-équivalent. On pourra utiliser un langage de programmation dont on prouvera que chaque instruction est calculable par une machine de TURING et dont la succession de deux actions est elle-même calculable.

On choisit ici d'utiliser le Haskell une fois de plus. Cette fois l'argument principal sera constitué par la puissance des ordinateurs réels utilisant uniquement des instructions assembleur proches de celles des machines RAM (cf. Annexe). Ainsi un ordinateur réel ne peut jamais dépasser la puissance de calcul d'une machine de TURING.

On implémente ainsi un λ-terme. On utilisera les variables sous forme de chaines de caractères.

```haskell
data Lambda_terme a =    App (Lambda_terme a) (Lambda_terme a) --L'application
                        |Abs (a) (Lambda_terme a) --L'abstraction
                        |Var a    --La variable
                        deriving Show
```

On commence par donner des fonctions pour gérer les listes.

```
split[]=([],[])
split[x]=([x],[])
split (s:t:q)= (s:a,t:b)
                where (a,b)=split q

merge l []=l
merge [] l=l
merge (t1:q1) (t2:q2)
    |t1<=t2 = t1:(merge q1 (t2:q2))
    |otherwise = t2:(merge (t1:q1) q2)

mSort []=[]
mSort [x]=[x]
mSort l=merge (mSort c)(mSort d)
           where (c,d)=split l

union2 [] l=l
union2 l []=l
union2 (t1:q1) (t2:q2)
    |t1==t2 = t1:(union2 q1 q2)
    |t1<t2 = t1:(union2 q1 (t2:q2))
    |t1>t2 = t2:(union2 (t1:q2) q2)

difference2 [] l=[]
difference2 l []=l
difference2 (t1:q1) (t2:q2)
    |t1==t2 = difference2 q1 q2
    |t1<t2 = t1:(difference2 q1 (t2:q2))
    |t1>t2 = difference2 (t1:q1) q2

doublon []=[]
doublon [a]=[a]
doublon (a:b:q)
    |a==b = doublon (a:q)
    |otherwise = a:(doublon (b:q))

union l1 l2=
    union2 (doublon (mSort l1)) (doublon(mSort l2))

difference l1 l2=
    difference2 (doublon (mSort l1)) (doublon(mSort l2))

appartient a []=False
appartient a (t:q)
    |a==t = True
    |otherwise = appartient a q
```

On peut ainsi facilement donner les codes pour former l'ensemble des variables libres, faire la substitution et l'évaluation.

```
freeVars (App a b)= union (freeVars a) (freeVars b)
freeVars (Abs a b)= difference (freeVars b) [a]
freeVars (Var a)  = [a]

sub m x (Var y)
    |x==y = m
    |otherwise = Var y

sub m x (App u v) = App (sub m x u) (sub m x v)

sub m x (Abs y e)
    |x==y = Abs y e
    |otherwise = Abs y (sub m x e)

eval2 (Var x) = Var x
eval2 (App (Abs x e) u) = sub u x (eval e)
eval2 (Abs x e) = Abs x (eval e)
eval2 (App u v) = App (eval u) (eval v)

eqLambdaTerme (Var a) (Var b)= a==b
eqLambdaTerme (App u v) (App a b)= (eqLambdaTerme u a) && (eqLambdaTerme v b)
eqLambdaTerme (Abs x a) (Abs y b)= (x==y) && (eqLambdaTerme a b)
eqLambdaTerme _ _= False

eval t = let w=(eval2 t) in
         if eqLambdaTerme w t then
             t
         else
             eval w
```

On peut définir quelques λ-termes en exemples

```
zero=Abs "f" (Abs "x" (Var "x"))
succ=Abs "n" (Abs "g" (Abs "y" (App (Var "g") (App (App (Var "n") (Var "g")) (Var "y")))))
un=eval(App Main.succ zero)
deux=eval(App Main.succ un)
true=Abs "a" (Abs "b" (Var "a"))
false=Abs "c" (Abs "d" (Var "d"))
isZero=Abs "k" (App (App (Var "k") (Abs "x" false)) (true))
```

et avoir la session suivante :

```
*Main> eval (App isZero zero)
Abs "a" (Abs "b" (Var "a"))
*Main> eval (App isZero un)
Abs "c" (Abs "d" (Var "d"))
*Main> eval (App Main.succ un)
Abs "g" (Abs "y" (App (Var "g") (App (Var "g") (Var "y"))))
```

10.4 Conclusion

Théorème 7 (TURING-équivalence du λ-calcul)**.**

La classe des fonctions calculables selon le λ-calcul est égale à la classe des fonctions calculables pour les machines de TURING.

Démonstration. $(\Rightarrow)$: On a prouvé qu'avec un système équivalent au λ-calcul, ici le Haskell, on pouvait simuler des machines de TURING. Pour cela on a explicité un programme qui simule les machines de TURING.

$(\Leftarrow)$: On a explicité une façon de réduire un λ-terme à l'aide d'une machine de TURING. Ainsi les machines de TURING ne sont pas moins puissantes qu'un λ-terme.

La classe des fonctions calculables selon le λ-calcul est égale à la classe des fonctions calculables pour les machines de TURING. □

Chapitre 11

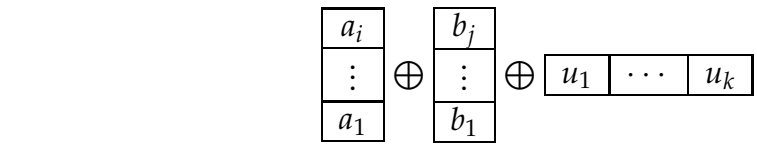
TURING-équivalence des machines à deux piles

On va ici montrer l'équivalence entre une machine de TURING et une machine à 2 piles. Le principe de la démonstration est d'utiliser la ressemblance des définitions des fonctions de transition. On utilise aussi une décomposition adéquate du ruban en deux piles.

11.1 Simulation des machines à deux piles par une machine de TURING

Soit $(R, r_0, \Lambda, \mathcal{Z}, z_0, F, \zeta)$ un automate à deux piles. On va construire une machine de TURING $(Q, \Gamma, B, \Sigma, q_0, \delta, F)$, notée $\mathcal{M}$, qui simule l'exécution de $(R, r_0, \Lambda, \mathcal{Z}, z_0, F, \zeta)$.

On combine les deux piles et le mot en un seul ruban de la façon suivante (en supposant $i < j$) :



$\Leftrightarrow$

0	u_1	$\cdots$	u_k	$\diamondsuit$	$\diamondsuit$	a_1	b_1	$\cdots$	a_i	b_i	0	b_{i+1}	$\cdots$	0	b_j	0

On va définir une machine de TURING grâce à des tables abrégées et on ne donnera donc pas la description explicite de la machine.

On crée d'abord des fonctions permettant de se placer correctement.

m-config.	symbole	opérations	m-config rés.
$at\_begin\,(\mathcal{E})$	0	G	$at\_begin_1\,(\mathcal{E})$
	quelconque	G	$at\_begin\,(\mathcal{E})$
$at\_begin_1\,(\mathcal{E})$	0	DD	$\mathcal{E}$
	quelconque	G	$at\_begin\,(\mathcal{E})$

$at\_begin(\mathcal{E})$ place la tête de lecture sur la première lettre du ruban puis $\to \mathcal{E}$.

m-config.	symbole	opérations	m-config rés.
$on\_stack_1\,(\mathcal{E})$			$at\_begin(st_1(\mathcal{E}))$
$st_1\,(\mathcal{E})$	$\diamondsuit$		$\mathcal{E}$
	quelconque	D	$st_1(\mathcal{E})$
$on\_stack_2\,(\mathcal{E})$			$st_1\,(r\,(\mathcal{E}))$

$on\_stack_1(\mathcal{E})$ place la tête de lecture sur le premier $\diamondsuit$. $on\_stack_2(\mathcal{E})$ la place sur le second. Ceci est justifié par le fait que les deux $\diamondsuit$ sont chacun dans la continuité d'une pile et en forment le fond. Il suffit alors de parcourir le ruban en sautant une case sur deux jusqu'à trouver un 0 pour atteindre le bout de la pile.

On peut alors écrire la table de transition de $\mathcal{M}$. $(R, r_0, \Lambda, \mathcal{Z}, z_0, F, \zeta)$. On suppose $R = \{r_0, \ldots, r_{n-1}\}$, $\Lambda = \{u_1, \ldots, u_{m-1}\}$, $\mathcal{Z} = \{z_0, \ldots, z_{p-1}\}$ où $n = |R|$, $m = |\Lambda|$ et $p = |\mathcal{Z}|$.

On commence avec l'état q_0 et sur la case contenant le premier $\diamondsuit$. Le but est de récupérer successivement les symboles au sommet des piles puis la lettre à lire et enfin d'empiler les nouveaux symboles. On revient après au premier $\diamondsuit$ puis on passe dans l'état correspondant à celui de la machine à deux piles.

On considère la transition $(q, a, (z_1, z_2)) \mapsto (r, (y, y'))$.

Si $z_1 = z_0 : q \to q_{z_0}$

Si $z_1 \neq z_0$:

m-config.	symbole	opérations	m-config rés.
q	0	GG	$\overline{q}$
	quelconque	DD	q
$\overline{q}$		E	$on\_stack_2(q_\#)$

A ce moment le symbole z_1 est récupéré. On va maintenant lire le symbole de la seconde pile :

Si $z_2 = z_0 : q_{z_1} \to q_{z_1,z_0}$

Si $z_2 \neq z_0$:

m-config.	symbole	opérations	m-config rés.
q_{z_1}	0	GG	$\overline{q_{z1}}$
	quelconque	DD	q_{z_1}
$\overline{q_{z_1}}$		E	$on\_begin(q_{z_1,\#})$

On va maintenant lire une lettre du mot en entrée.
Si $a = \varepsilon : q_{z_1,z_2} \rightarrow q_{z_1,z_2,\varepsilon}$
Si $a \neq \varepsilon$:

m-config.	symbole	opérations	m-config rés.
q_{z_1,z_2}		E	$q_{z_1,z_2,\#}$

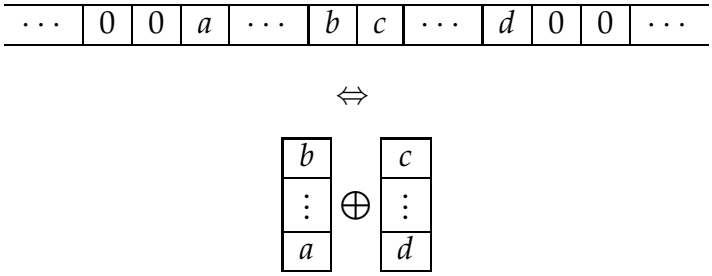
Une fois qu'on a récupéré les trois éléments on effectue la transition avant de replacer les résultats aux bons endroits.
$q_{z_1,z_2,a} \rightarrow on\_stack_1(\dot{r}_{y,y'})$

m-config.	symbole	opérations	m-config rés.
$\dot{r}_{y,y'}$	0	Iy	$on\_stack_2(\dot{r}_{y'})$
	quelconque	DD	$\dot{r}_{y,y'}$
$\dot{r}_{y'}$	0	Iy'	$on\_stack_1(r)$
	quelconque	DD	$\dot{r}_{y'}$

11.2 Simulation des machines de TURING par une machine à deux piles

Soit $M = (Q, \Gamma, B, \Sigma, q_0, \delta, F)$ une machine de TURING. On va construire un automate à deux piles $(R, r_0, \Lambda, \mathcal{Z}, z_0, G, \zeta)$ qui simule l'exécution de M.

En remarquant qu'on a (si le b est le symbole sous la tête de lecture) la décomposition du ruban suivante :



On suppose que $Q = \{q_0, \ldots, q_{n-1}\}$, $\Gamma = \{a_0, \ldots, a_{p-1}\}$, $F = \{q_{i_1}, \ldots, q_{i_m}\}$ et $B = a_0$ où $n = |Q|$ et $p = |\Gamma|$.

On pose $\mathcal{Z} = \Gamma$. D'autre part on construit un ensemble d'états de cardinal $2n$: $R = \{r_0, r'_0, \ldots, r_{n-1}, r'_{n-1}\}$ et $G = \{r_{i_1}, \ldots, r_{i_m}\}$.

Le but est de coder chaque transition de la machine de TURING par des transitions de la machine à deux piles. Pour cela, on distingue différents cas possibles de transitions de la machine de TURING selon le mouvement effectué :

— si $(q, a_i) \mapsto (q', a_j, \longleftarrow)$, on construit $(r, \varepsilon, (a_i, z_0)) \mapsto (r', (z_0, a_j))$.

— si $(q, a_i) \mapsto (q', a_j, \bigcirc)$, la transition devient $(r, \varepsilon, (a_i, z_0)) \mapsto (r', (a_j, z_0))$.

— si $(q, a_i) \mapsto (q', a_j, \longrightarrow)$, on crée 2 transitions. La première permet d'écrire le nouveau symbole $(r, \varepsilon, (a_i, z_0)) \mapsto (r'', (a_j, z_0))$, puis la seconde assure le décalage du ruban $(r'', \varepsilon, (z_0, a_k)) \mapsto (r', (a_k, z_0))$.

11.3 Conclusion

Théorème 8 (TURING-équivalence des automates à deux piles)**.**

La classe des fonctions calculables par les automates à deux piles est égale à la classe des fonctions calculables pour les machines de TURING.

Démonstration. ($\Rightarrow$) : On a simulé une machine à deux piles grâce à une machine de TURING ce qui prouve que les automates à deux piles ne sont pas plus puissants que les machines de TURING.

($\Leftarrow$) : Réciproquement on a simulé les transitions et donc l'exécution d'une machine de TURING par un automate à deux piles. Ainsi les machines de TURING ne sont pas plus puissantes que les automates à deux piles.

La classe des fonctions calculables selon les automates à deux piles est égale à la classe des fonctions calculables pour les machines de TURING. □

Chapitre 12

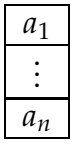
Équivalence des machines à plusieurs piles et des machines à plusieurs compteurs

Le but est de montrer l'équivalence entre des machines à compteurs et les machines multipiles. Pour cela on procédera de manière constructive en se donnant la machine à simuler et en décrivant la machine simulant.

12.1 Simulation des machines à piles grâce aux machines à compteurs

On se donne une machine à k piles $(Q, q_0, \Sigma, \mathcal{Z}, z_0, F, \delta)$. Quitte à supposer les piles non vides à l'état initial et quitte à rajouter une pile contenant le mot d'entrée on peut supposer que le mot d'entrée est vide.

On peut supposer que $\Sigma = [\![0, r-1]\!]$ et considérer une pile comme un mot $w \in [\![0, r-1]\!]^\star$. Ainsi la pile



peut être représentée par l'entier $i = \sum_{k=1}^{n} a_k r^{k-1}$.

Le but est de se servir d'une machine à $k+1$ compteurs. Les k premiers compteurs stockent les piles. Le dernier sert au calcul. On montrera par la suite que toute machine à $k+1$ compteurs peut être simulée par une machine à 2

compteurs. On rappelle aussi que la décrémentation à partir de la valeur nulle donne 0 pour une machine à compteurs.

On peut ainsi créer des fonctions de manipulation de piles :

— Dépiler : $i \to i \operatorname{div} r$

— Empiler a : $i \to i \times r + a$

— Lire le sommet de la pile : $i \mod r$

On détaille la fonction qui permet de dépiler. On note C un compteur supplémentaire.

```
C=0;
while(i!=0)
{
    i=i-r;
    C=C+1;
}
while(C!=0)
{
    C=C-1;
    i=i+1;
}
```

On implémente ainsi la fonction qui empile.

```
C=0;
while(i!=0)
{
    i=i-1;
    C=C+r;
}
C=C+a;
while(C!=0)
{
    C=C-1;
    i=i+1;
}
```

On va maintenant s'intéresser à la fonction de lecture du sommet de la pile. On doit pour cela d'abord se convaincre qu'on peut comparer une variable avec un nombre fixé à l'avance, en l'occurrence r. Le but est de comparer i avec 0, ce qui est faisable, puis de retirer 1 si i est non nul. On itère au plus r fois en s'arrêtant avant si i s'annule plus tôt. Dans ce dernier cas, on sait que $i < r$. Si à la fin des r itérations $i = 0$, i valait initialement r. Dans le dernier cas (i non nul à la fin) on sait que i était supérieur à r. Dans tous les cas, on peut restaurer i à la fin.

```
C=i;
while(C>=r)
{
    C=C-r;
}
```

À la fin du procédé, C contient $i \mod r$.

12.2 Simulation des machines à compteurs à l'aide des machines à piles

On va ici simplement simuler chacune des instructions possibles indépendamment les unes des autres. On se donne une machine à k compteurs que l'on va simuler avec une machine à k piles. Chacune des piles ne va contenir que des copies du même symbole. La taille de la pile correspond à la valeur dans le compteur correspondant. Au plus profond de chaque pile, on place un symbole particulier marquant le fond de la pile.

Pour simuler une incrémentation, on empile une nouvelle fois le symbole, pour la décrémentation, on en retire un si possible. Le saut conditionnel demande de vérifier si la pile est vide, ce qui ne pose aucun problème puisqu'on a un symbole marquant le fond de la pile.

Il est ainsi possible et facile de simuler une machine à compteurs par une machine à piles.

12.3 Conclusion

Théorème 9.

La classe des fonctions calculables grâce aux machines multipiles est égale à la classe des fonctions calculables grâce aux machines à compteurs.

Démonstration. On a montré que ces deux modèles de calcul peuvent se simuler mutuellement.

La classe des fonctions calculables pour les machines à plusieurs piles est donc égale à la classe des fonctions calculables pour les machines à compteurs. □

Chapitre 13

Équivalence des machines à plusieurs compteurs et des machines à deux compteurs

Cette étape est particulièrement importante, bien que relativement simple. Elle permet en effet de prouver que les machines à deux compteurs sont équivalentes aux automates à piles multiples et ainsi aux machines de TURING. L'autre intérêt réside dans le fait que ce système a une description simple. Il est ainsi aisé de montrer qu'un système est TURING-complet en lui faisant simuler une machine à deux compteurs.

13.1 Simulation des machines à compteurs grâce aux machines à deux compteurs

On sait qu'une machine de TURING est équivalente à une machine à deux piles. On va ainsi simuler uniquement les machines à trois compteurs puisqu'on sait qu'elles sont équivalentes à deux piles.

On note respectivement i,j et k les valeurs des trois compteurs de la machine à simuler. On note C_1 et C_2 les deux compteurs de la machine qui permet de simuler les trois compteurs. Le principe est de stocker $m = 2^i \times 3^j \times 5^k$ dans C_1. D'autre part C_2 va être utilisé selon une méthode connue pour faire les calculs sur C_1.

Pour s'épargner une réécriture de toutes les fonctions, on remarquera que ces fonctions sont très similaires au cas des machines à $k+1$ compteurs qui simulent les machines à k piles. On peut alors adapter légèrement ces implémentations pour donner les fonctions utilisables ici.

13.2 Simulation des machines à deux compteurs à l'aide des machines à compteurs

On peut simuler une machine à deux compteurs grâce à une machine à plusieurs compteurs en n'utilisant pas les compteurs supplémentaires. En effet les machines à deux compteurs ne sont qu'une restriction des machines à compteurs générales.

13.3 Conclusion

Théorème 10.

La classe des fonctions calculables grâce aux machines à plusieurs compteurs est égale à la classe des fonctions calculables grâce aux machines à deux compteurs.

Démonstration. On a montré que les machines à deux compteurs peuvent simuler trois compteurs. D'autre part les machines à trois compteurs ne sont qu'une restriction des machines à plusieurs compteurs.

La classe des fonctions calculables pour les machines à plusieurs compteurs est donc égale à la classe des fonctions calculables pour les machines à deux compteurs. □

Chapitre 14

Équivalence des autres modèles de calcul

Les autres démonstrations sont particulièrement délicates à faire par équivalences. On utilisera donc une suite d'implications induisant l'équivalence entre tous les modèles présentés. La méthode générale consiste à prouver grâce à des chemins fermés l'équivalence des modèles de calcul restants, puis on utilisera l'équivalence des machines de TURING et des machines à deux piles pour conclure sur la TURING-équivalences des modèles restants.

14.1 Simulation des machines à compteurs grâce à des RAM

On va dans un premier temps prouver qu'une machine RAM peut simuler n'importe quelle machine à compteurs.

On se donne une machine à k compteurs que l'on va simuler grâce à une machine RAM. Le $i^{\text{ème}}$ compteur sera simulé par $i^{\text{ème}}$ registre de la machine RAM, pour $i \in entiers1k$.

Il est nécessaire pour cela de simuler chaque instruction possible.

On trouve ainsi le code permettant l'incrémentation du compteur i :

```
   0 load i
1 incr
2 store i
```

Celui permettant la décrémentation du même compteur :

```
0 load i
1 decr
2 store i
```

Ainsi que le saut conditionnel sur le compteur i :

```
0 load i
1 jz (label 1)
2 jump (label 2)
```

On peut donc coder chacune des instructions des machines à compteurs grâce aux instructions des machines RAM. Par conséquent les machines RAM sont plus puissantes (au sens large) que les machines à compteurs.

14.2 Simulation des RAM à l'aide des machines de TURING à plusieurs rubans

L'idée de la simulation est relativement simple. La machine RAM est simulée par une machine de TURING à deux bandes. La première bande contient le contenu de tous les registres de la machine RAM y compris l'accumulateur. Les entiers contenus dans les registres sont écrits en binaire sur l'alphabet $\{0, 1\}$. Le contenu de l'accumulateur est mis en premier sur la bande, suivi du contenu du registre R_1, puis du registre R_2 et ainsi de suite. Les écritures binaires des contenus sont séparées par le caractère ','. À un instant donné, la bande ne contient que les valeurs d'un nombre fini de registres. Les contenus des autres sont par convention leur valeur initiale qu'on suppose être zéro.

À chaque instruction du programme, on va associer une partie de la machine de TURING qui sera chargée de simuler le fonctionnement de l'instruction donnée. Ces parties sont ensuite mises bout à bout pour constituer une machine de TURING globale.

Les manipulations de registres nécessitent de savoir retrouver le contenu d'un registre. Pour ce faire, la machine de TURING écrit le numéro du registre sur la seconde bande. Elle parcourt ensuite la première bande en incrémentant l'entier sur la seconde bande à chaque passage sur le séparateur ',' sur la première bande. Si la machine ne trouve pas assez de virgules sur la première bande, elle en ajoute en les séparant par 0 qui est le contenu initial d'un registre. Ensuite, les manipulations consistent à recopier le contenu du registre dans l'accumulateur pour `load` ou le contenu de l'accumulateur dans le registre pour `store`. Dans le cas d'une indirection (adressage indirect), la machine copie d'abord le contenu du registre sur la seconde bande puis recherche à nouveau le registre correspondant avant de copier les contenus.

Simuler les instructions arithmétiques est facile puisqu'il s'agit simplement de modifier le contenu de l'accumulateur qui se trouve au début de la première bande.

Les instructions de saut ne sont pas vraiment simulées. Elles influent sur la façon dont les différentes parties sont combinées pour former la machine de TURING globale.

Les machines de TURING à plusieurs rubans sont donc plus puissantes (au sens large) que les machines RAM.

14.3 Simulation des MTTM grâce aux machines à piles

On a déjà montré qu'il est possible de simuler une machine de TURING grâce aux automates à deux piles. On veut alors prouver qu'il est possible de simuler une machine de TURING à k rubans grâce à une machine à $2k$ piles. Les mémoires sont équivalentes puisqu'il est connu qu'un ruban est équivalent à deux piles.

Il ne reste plus qu'à adapter très légèrement la fonction de transition pour simuler la machine de TURING par la machine à plusieurs piles.

14.4 Simulation des machines à deux piles à l'aide des machines à piles

Les automates à deux piles n'étant qu'une restriction des automates à plusieurs piles, il est possible de simuler les automates à deux piles avec un automate qui aurait un nombre de piles supérieur. En effet, il suffit pour cela de restreindre l'utilisation à deux piles et de ne pas utiliser les piles restantes.

14.5 Simulation des machines à deux compteurs à l'aide des machines à deux piles

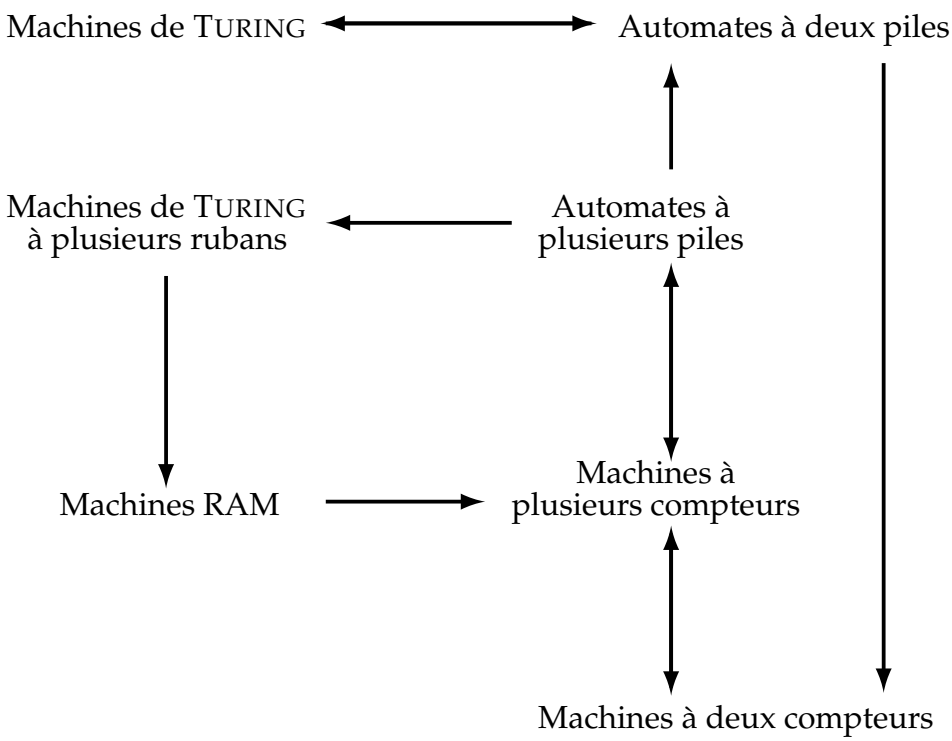
La démonstration est la même que pour le cas général déjà évoqué. Il suffit de stocker dans chaque pile un caractère pour en marquer le fond, puis une quantité de symboles identiques pour coder la valeur du compteur.

14.6 Conclusion

Théorème 11.

Les automates à piles, les machines à compteurs, les machines RAM et les machines de TURING à plusieurs rubans sont TURING-équivalentes.

Démonstration. On peut résumer toutes ces implications par un graphe dont on voit le caractère eulérien prouvant ainsi l'équivalence de tout ces modèles :



On a ainsi la TURING-équivalence de tous ces modèles de calcul.

□

Troisième partie

La calculabilité

Chapitre 15

Préliminaires

La notion de calculabilité est une notion intuitive en mathématiques. Dans les années 1930, des logiciens comme GÖDEL, CHURCH ou TURING ont cependant cherché à la formaliser. Il s'agit ici de trouver la classe des fonctions qui peuvent être considérées comme calculables.

On ne considère ici que des fonctions de $\mathbb{N}$ dans $\mathbb{N}$. Intuitivement, une fonction f définie sur un ensemble E est calculable s'il existe un algorithme permettant d'obtenir pour tout x de E son image $f(x)$ en un nombre fini d'étapes.

Par conséquent il semble raisonnable de définir également des nombres calculables. Intuitivement, ce serait ceux qu'une fonction permettrait de calculer. Plus précisément, ce sont les nombres pour lesquels il existe une fonction qui calcule chaque décimale en un nombre fini d'étape.

15.1 Définition

Définition 39 (Calculabilité).

Les fonctions semi-calculables (ou récursives partielles) sont les fonctions calculées par une machine de TURING (ou tout système équivalent).

Définition 40.

Les fonctions calculables sont les fonctions semi-calculables totales.

On remarque que la terminologie de fonction récursive correspond à la notion de fonction récursive définie précédemment puisque les machines de TURING sont équivalentes aux fonctions récursives.

Lorsqu'on définit la notion de calculabilité, la thèse de CHURCH-TURING joue un rôle important. Celle-ci postule l'égalité entre les fonctions calculables au sens de l'intuition et les fonctions calculées par machine de TURING. Elle avance aussi l'égalité entre les fonctions calculées par machine de TURING et toute méthode effective de calcul.

Une méthode effective de calcul doit remplir les conditions suivantes :

— l'algorithme consiste en un ensemble fini d'instructions simples et précises qui sont décrites avec un nombre limité de symboles,
— l'algorithme doit toujours produire le résultat en un nombre fini d'étapes,
— l'algorithme peut en principe être suivi par un humain avec seulement du papier et un crayon,
— l'exécution de l'algorithme ne requiert pas d'intelligence de l'humain sauf celle qui est nécessaire pour comprendre et exécuter les instructions.

15.2 Propriétés des fonctions calculables

Proposition 32.

L'ensemble des fonctions semi-calculables est dénombrable.

Démonstration. On va le démontrer en dénombrant les machines de TURING.

Étant donné un nombre fini n d'états, p de symboles, il existe $(3np)^{np}$ fonctions de transitions. On peut identifier toute machine de TURING à n états et à p symboles aux machines dont l'ensemble des états est $[\![0, n-1]\!]$ et l'ensemble des symboles est $[\![0, p-1]\!]$ et où le caractère blanc est 0. Ensuite il faut prendre $\Sigma \subseteq \Gamma \setminus \{B\}$, ce qui constitue 2^{p-1} choix, q_0 soit n choix et $F \subseteq Q$ soit 2^n choix. Donc le cardinal de l'ensemble $\mathbb{M}(n,p)$ des machines à n états et p symboles est :

$$|\mathbb{M}(n,p)| = 2^{p-1+n} n (3np)^{np}$$

Le nombre de machines de TURING pour n états et à p symboles est donc fini. Or $\mathbb{N}^2$ est dénombrable donc l'ensemble $\left\{\mathbb{M}(n,p) \;\middle|\; (n,p) \in \mathbb{N}^2\right\}$ des machines de TURING à n états et p symboles est donc dénombrable. Donc le nombre de fonctions calculables au sens de TURING est dénombrable.

Il en est de même pour les autres modèles de calcul. □

On peut raisonner de même pour les autres modèles à automates. Les autres demandent une preuve inductive.

15.3 Les fonctions incalculables

Proposition 33.

L'ensemble des fonctions de $\mathbb{N}$ dans $\mathbb{N}$ est indénombrable et en bijection avec $\mathbb{R}$.

Démonstration. On montre d'abord qu'il existe une surjection de $\mathbb{N}^{\mathbb{N}}$ dans $\mathbb{R}$. Pour toute partie $A \in \mathcal{P}(\mathbb{N})$, on appelle φ_A :

$$\begin{aligned} \varphi_A : \mathbb{N} &\to \{0,1\} \\ x &\mapsto [x \in A] \end{aligned}$$

On appelle $\mathfrak{F}$, l'ensemble des suites naturelles constituées uniquement de 0 et de 1.

$$\mathfrak{F} = \left\{ (u_n)_{n\in\mathbb{N}} \in \mathbb{N}^{\mathbb{N}} \;\middle|\; \forall n \in \mathbb{N}, u_n \in \{0,1\} \right\}$$

On a $\mathfrak{F} \subsetneq \mathbb{N}^{\mathbb{N}}$.
On définit également $\mathcal{T}$:

$$\begin{aligned} \mathcal{T} : \mathcal{P}(\mathbb{N}) &\to \{0,1\}^{\mathbb{N}} \\ A &\mapsto \varphi_A \end{aligned}$$

et Ψ :

$$\begin{aligned} \Psi : \{0,1\}^{\mathbb{N}} &\to \mathfrak{F} \\ \varphi_A &\mapsto (\varphi_A(n))_{n\in\mathbb{N}} \end{aligned}$$

On sait que $\mathcal{T}$ est bijective. Il est clair que Ψ est surjective. Montrons qu'elle est injective. Soit $\varphi_A, \varphi_B \in \{0,1\}^{\mathbb{N}}$ et supposons que $\Psi(\varphi_A) = \Psi(\varphi_B)$.On a alors :

$$\begin{aligned} \Psi(\varphi_A) = \Psi(\varphi_B) &\Leftrightarrow (\varphi_A(n))_{n\in\mathbb{N}} = (\varphi_B(n))_{n\in\mathbb{N}} \\ &\Leftrightarrow \forall n \in \mathbb{N}, \varphi_A(n) = \varphi_B(n) \\ &\Leftrightarrow \varphi_A = \varphi_B \end{aligned}$$

Ψ est donc bien injective donc bijective. Ce qui implique que $\mathcal{T} \circ \Psi$ est bijective. $\mathfrak{F}$ est donc en bijection avec $\mathcal{P}(\mathbb{N})$, donc avec $\mathbb{R}$, donc $\mathfrak{F}$ est indénombrable. Or $\mathfrak{F} \subsetneq \mathbb{N}^{\mathbb{N}}$, donc $\mathbb{N}^{\mathbb{N}}$ il existe une surjection de $\mathfrak{F}$ dans $\mathbb{R}$, donc $\mathbb{N}^{\mathbb{N}}$ est indénombrable. Donc l'ensemble des fonctions est indénombrable.

On montre maintenant qu'il existe une injection de $\mathbb{N}^{\mathbb{N}}$ dans $\mathbb{R}$.

On définit l'application $\mathfrak{E}$ par :

$$\mathfrak{E} : \mathbb{N}^{\mathbb{N}} \hookrightarrow \mathcal{P}\left(\mathbb{N}^2\right)$$
$$(u_n)_{n\in\mathbb{N}} \mapsto \left\{(i, u_i) \in \mathbb{N}^2 \,\middle|\, i \in \mathbb{N}\right\}$$

Soit $a, b \in \mathbb{N}^{\mathbb{N}}$. On suppose que $a \neq b$.
Il existe N tel que $a_N \neq b_N$. On a $(N, a_N) \in \mathfrak{E}(a)$ et $(N, b_N) \in \mathfrak{E}(b)$.
De plus $\forall u \in \mathbb{N}^{\mathbb{N}}, \forall k \in \mathbb{N}, \exists! i \in \mathbb{N} : (k, i) \in \mathfrak{E}(u)$.
Donc $(N, a_N) \notin \mathfrak{E}(b)$ et $(N, b_N) \notin \mathfrak{E}(a)$ ce qui implique :

$$\mathfrak{E}(a) \neq \mathfrak{E}(b)$$

Ce qui justifie l'injectivité de $\mathfrak{E}$.

Or $\mathbb{N}$ est en bijection avec $\mathbb{N}^2$ et $\mathcal{P}(\mathbb{N})$ est en bijection $\mathbb{R}$. Donc $\mathcal{P}\left(\mathbb{N}^2\right)$ est en bijection avec $\mathbb{R}$. Donc il existe une injection de $\mathbb{N}^{\mathbb{N}}$ dans $\mathbb{R}$.

On a montré qu'il existe une surjection et une injection de $\mathbb{N}^{\mathbb{N}}$ dans $\mathbb{R}$ donc, d'après le théorème de CANTOR-BERNSTEIN, ces deux ensembles sont en bijection.

□

Proposition 34.

Il existe des fonctions non calculables.

Corollaire 5.

L'ensemble des fonctions incalculables est indénombrable.

Démonstration. Les ensembles des fonctions calculables et des fonctions incalculables réalisent une partition de l'espace des fonctions. Or les fonctions calculables sont dénombrables donc l'ensemble des fonctions incalculables est indénombrable. □

Exemple 19 (Problème de l'arrêt)**.**

Il n'existe pas d'algorithme H qui prendrait en entrée un algorithme A et l'entrée x pour A, et déterminerait si A s'arrête après un nombre fini d'étapes.

Exemple 20 (Castor affairé).

On cherche ici à déterminer une machine de TURING à n états opérant sur un alphabet de 0 et de 1 qui écrive un maximum de 1 possible avant de s'arrêter. Les machines de TURING utilisées ici sont supposées s'arrêter (pas de machine écrivant une infinité de 1...).
L'objectif est le suivant : la fonction qui détermine le maximum de 1 pouvant être écrit par une machine de TURING à n états est-elle calculable ?

Définition 41.

On considère un alphabet de bande de taille 2, comprenant des 1 et des 0. Soit E_n l'ensemble fini non vide des machines de TURING à n états et 2 symboles qui finissent par s'arrêter, $\varphi(M)$ le nombre de 1 écrit avant l'arrêt de M, Σ est la fonction du castor affairé définie sur $\mathbb{N}^*$ par :

$$\Sigma(n) = \max\{\varphi(M) \mid M \in E_n\}$$

Proposition 35.

Σ est strictement croissante.

Démonstration. Soit $n \in \mathbb{N}^*$ et $\mathcal{M} \in E_n$ tel que $\varphi(\mathcal{M}) = \Sigma(n)$. On note q_f l'état final de $\mathcal{M}$ lors de son exécution sur le ruban vide. On complète $\mathcal{M}$ en une machine $\mathcal{N} \in E_{n+1}$ qui a une état supplémentaire noté $q_\mathcal{N}$ et des transitions supplémentaires suivantes :

$$\begin{aligned}(q_f, 0) &\mapsto (q_\mathcal{N}, 1, HALT)\\ (q_f, 1) &\mapsto (q_\mathcal{N}, 1, \leftarrow)\\ (q_\mathcal{N}, 0) &\mapsto (q_\mathcal{N}, 1, HALT)\\ (q_\mathcal{N}, 1) &\mapsto (q_\mathcal{N}, 1, \leftarrow)\end{aligned}$$

On a clairement $\varphi(\mathcal{N}) = \Sigma(n) + 1$ et $\varphi(\mathcal{N}) \leqslant \Sigma(n+1)$ d'où

$$\Sigma(n+1) > \Sigma(n)$$

Donc Σ est strictement croissante. □

Remarque 6.

On a, par exemple :

$$\Sigma(1) = 1$$
$$\Sigma(2) = 4$$
$$\Sigma(3) = 6$$

Proposition 36.

Σ n'est pas calculable.

Démonstration. On procède par l'absurde.

On suppose que Σ est calculable.

Alors il existe une machine $\mathcal{M}_\Sigma$ qui calcule Σ en prenant un entier écrit en binaire sur son ruban. On note m son nombre d'états.

De plus, pour tout $n \in \mathbb{N}^*$, il existe une machine qui a au plus n états qui écrit n en binaire sur le ruban. On la note $\mathcal{M}_n$

On peut aussi créer une machine de TURING qui multiplie par deux un nombre binaire sur le ruban. Il suffit en effet d'ajouter un 0 au début du nombre. On l'appelle $\mathcal{M}_p$ et on note k son nombre d'états.

Enfin il existe une machine qui, étant un nombre n en binaire sur son ruban, écrit n 1 consécutifs. C'est la fonction de duplication. On la note $\mathcal{M}_d$ et on note p son nombre d'état.

Aussi, $\mathcal{M}_d \circ \mathcal{M}_\Sigma \circ \mathcal{M}_p \circ \mathcal{M}_n(\varepsilon) = d_{2n}(1)$ et $\mathcal{M}_d \circ \mathcal{M}_\Sigma \circ \mathcal{M}_p \circ \mathcal{M}_n$ a au plus $n + k + m + p$ états. D'où $\Sigma(n + k + m + p) \geqslant \Sigma(2n)$

Donc d'après la monotonie stricte de Σ, on déduit immédiatement que $\forall n \in \mathbb{N}, n > m + k + p \Rightarrow \Sigma(n + k + m + p) < \Sigma(2n)$.

D'où $\forall n > m + k + p, \Sigma(2n) \leqslant \Sigma(n + k + m + p) < \Sigma(2n)$.

On a une contradiction donc Σ n'est pas calculable. □

15.4 Les nombres calculables

On peut comprendre alors qu'il est pertinent de définir un ensemble de nombres calculables. Intuitivement ce sont des nombres dont le calcul peut s'effectuer de façon algorithmique.

Définition 42.

Un nombre réel t est calculable s'il existe une machine de TURING qui écrit les décimales de t (par exemple sur les C-cases) dans une base arbitraire et qui ne s'arrête que si t a un nombre fini de décimales.

Notation 22.

On note $\mathbb{T}$ l'ensemble des nombres calculables.

Définition 43.

On appelle description d'un nombre calculable t, tout moyen de décrire sans ambiguïté t. En particulier, on peut donner explicitement le nombre, si son écriture est finie ou, sinon, une machine de TURING qui le calcule (ou sa description standard).

Définition 44.

Soit $m \in \mathbb{N}^*$.
Une fonction $f : \mathbb{R}^m \to \mathbb{R}$ est dite calculable s'il existe une machine de TURING qui calcule $f(t_1, \ldots, t_m)$ quand on lui donne en entrée une description des nombres calculables $t_1, \ldots, t_m$.

Définition 45.

Soit $(m, n) \in (\mathbb{N}^*)^2$.
Une fonction $f : \mathbb{R}^m \to \mathbb{R}^n$ est dite calculable si chaque fonction $p_{i,n} \circ f : \mathbb{R}^m \to \mathbb{R}^n$, pour $i \in [\![1, n]\!]$ est calculable.

On peut donc en particulier définir les fonctions de $\mathbb{C} \to \mathbb{C}$ calculables : ce sont celles dont les parties réelles et imaginaires sont calculables.

Proposition 37.

$\mathbb{T}$ est dénombrable.

Démonstration. En effet, il existe un nombre dénombrable de machines de TURING, $\mathbb{T}$ est donc au plus dénombrable. De plus, $\mathbb{N} \subseteq \mathbb{T}$ donc $\mathbb{T}$ est dénombrable. □

Proposition 38.

$\mathbb{Q} \subseteq \mathbb{T}$

Démonstration. Il s'agit de montrer qu'il existe une méthode de calcul effective permettant de calculer les décimales des nombres rationnels.

Une telle méthode existe nécessairement car il existe un algorithme de division d'où l'inclusion. □

Proposition 39.

On note $\mathbb{A}$, l'ensemble des nombres algébriques.
$\mathbb{A} \subseteq \mathbb{T}$

Démonstration. Il suffit en effet de pouvoir approcher les racines d'un polynôme avec un nombre de décimales exactes croissant, ce qui peut être fait par dichotomie.

En utilisant cette méthode il suffit en effet d'un nombre fini de données :

— Le polynôme à coefficient entiers
— Un intervalle contenant une unique racine (celle qu'on veut approximer) et tel que le polynôme y soit monotone, ce qui est possible en dérivant le polynôme si la racine est un extremum local. En effet, dans le cas contraire, il s'agit d'une racine multiple. On dérive alors le polynôme jusqu'à ce que la racine soit simple, le polynôme est alors monotone autour de ce point.

Il suffit ensuite de vérifier que le test de positivité d'un polynôme peut se faire en un nombre fini d'opérations. □

On va maintenant prouver qu'un certain nombre de fonctions usuelles sont calculables. L'idée générale est d'avoir une méthode itérative (typiquement une série) dont on est certain de la convergence et dont on va essayer de majorer l'erreur pour être sûr d'un nombre croissant de décimales.

Proposition 40.

La fonction

$$\begin{aligned} \exp : \mathbb{R} &\to \mathbb{R} \\ x &\mapsto e^x \end{aligned}$$

est calculable.

Démonstration. On sait que $e^x = \sum_{n=0}^{+\infty} \frac{x^n}{n!}$.

Or la suite des restes converge vers 0. Donc si les restes sont inférieurs à 10^{-n}, pour $n \in \mathbb{N}$, la somme partielle assure $n-2$ décimales exactes.

De plus, pour tout $n \in \mathbb{N}$:

$$\begin{aligned} \sum_{k=n}^{+\infty} \frac{x^k}{k!} &= \sum_{k=0}^{+\infty} \frac{x^{k+n}}{(k+n)!} \\ &= \frac{x^n}{n!} \sum_{k=0}^{+\infty} \frac{x^k}{(n+1)\cdots(n+k)} \\ &\leqslant \frac{x^n}{n!} \sum_{k=0}^{+\infty} \frac{x^k}{1\cdots k} \\ &\leqslant \frac{x^n e^x}{n!} \end{aligned}$$

D'autre part, on majore le reste par une puissance de 10.

$$\begin{aligned} \frac{x^n e^x}{n!} &\leqslant \frac{1}{10^{\varphi(n)}} \\ \frac{n!}{x^n e^x} &\geqslant 10^{\varphi(n)} \\ \ln\left(\frac{n!}{x^n e^x}\right) &\geqslant \varphi(n) \\ \ln(n!) - \ln(x^n e^x) &\geqslant \varphi(n) \\ \ln(n!) - n\ln(x) - x &\geqslant \varphi(n) \\ \sum_{i=1}^{n} \ln(i) - n\ln(x) - x &\geqslant \varphi(n) \end{aligned}$$

Or on a

$$\sum_{i=1}^{n} \ln\left(\frac{i}{x}\right) = n\left(\ln\left(\frac{n}{x}\right) - 1\right) + \frac{\ln(2\pi n)}{2} + \frac{1}{12n} - \frac{1}{360n^3} + \mathcal{O}\left(\frac{1}{n^5}\right)$$
$$\geqslant n(H_m(n) - \ln(x) - 1) + \frac{1}{2}H_m(n) + \frac{1}{12n} - \frac{1}{360n^3} + \mathcal{O}\left(\frac{1}{n^5}\right)$$

où

$$\begin{aligned} H_m : \mathbb{N} &\to \mathbb{Q} \\ n &\mapsto \sum_{i=1}^{n} \frac{1}{i} - 2 \end{aligned}$$

puisque $\forall n \in \mathbb{N}^*, \ln(n) > H_m(n)$.

On peut donc choisir $\varphi(n) := \max\left(n\left(H_m(n) - x - 1\right) + \frac{H_m(n)}{2}, C\right)$ où C est une constante qui majore le terme restant, indépendant de x. $C = 1$ suffit. Cette fonction est très clairement calculable. On a donc une minoration du nombre de décimal justes qu'on a à chaque étape. De plus $\lim_{n\to+\infty} \varphi(n) = +\infty$. Ainsi, on est certain de calculer toutes les décimales. Par conséquent, la fonction exponentielle est calculable. □

Corollaire 6.

$e \in \mathbb{T}$

Démonstration. Il s'agit juste de la fonction exp évaluée en 1. □

Corollaire 7.

La fonction

$$\begin{aligned} \sinh : \mathbb{R} &\to \mathbb{R} \\ x &\mapsto \sinh(x) \end{aligned}$$

est calculable.

Démonstration. On sait que $\sinh(x) = \frac{e^x - e^{-x}}{2}$ donc sinh est calculable. □

Corollaire 8.

La fonction

$$\begin{aligned} \cosh : \mathbb{R} &\to \mathbb{R} \\ x &\mapsto \cosh(x) \end{aligned}$$

est calculable.

Démonstration. On sait que $\cosh(x) = \frac{e^x + e^{-x}}{2}$ donc cosh est calculable. □

Corollaire 9.

La fonction

$$\begin{aligned} \sin : \mathbb{R} &\to \mathbb{R} \\ x &\mapsto \sin(x) \end{aligned}$$

est calculable.

Démonstration. On sait que

$$\forall x \in \mathbb{R}, \sin(x) = \sum_{i=0}^{+\infty} \frac{x^{2i+1}}{(2i+1)!}$$

donc $\forall x \in \mathbb{R}, \sin(x) = \frac{\sinh(ix)}{i}$. Or sinh est calculable donc sin est calculable. □

Corollaire 10.

La fonction

$$\begin{aligned} \cos : \mathbb{R} &\to \mathbb{R} \\ x &\mapsto \cos(x) \end{aligned}$$

est calculable.

Démonstration. On sait que

$$\forall x \in \mathbb{R}, \sin(x) = \sum_{i=0}^{+\infty} \frac{x^{2i}}{(2i)!}$$

donc $\forall x \in \mathbb{R}, \cos(x) = \cosh(ix)$. Or cosh est calculable donc cos est calculable. □

Proposition 41.

$\pi \in \mathbb{T}$

Démonstration. La fonction sin est strictement monotone sur $[3.1, 3.2]$. On a $\pi \in [3.1, 3.2]$ et $\sin(\pi) = 0$, par conséquent, on peut calculer π par dichotomie. □

En vérité la très large majorité des nombres raisonnablement définissables sont calculables. En effet la plupart des définitions mathématiques utilisent des fonctions calculables et fournissent ainsi un algorithme de calcul.

Proposition 42.

La réciproque de toute bijection calculable croissante est calculable.

Démonstration. Il suffit d'appliquer une dichotomie sur les valeurs de la fonction pour calculer la fonction réciproque. On sait que chaque étape de la dichotomie donne exactement une décimale en binaire. □

Corollaire 11.

La réciproque de toute bijection calculable monotone est calculable.

Démonstration. Immédiat avec la proposition précédente. □

Proposition 43.

La fonction

$$\begin{aligned} \ln : \mathbb{R} &\to \mathbb{R} \\ x &\mapsto \ln(x) \end{aligned}$$

est calculable.

Démonstration. ln a pour réciproque exp. Ces fonctions sont croissantes et exp est calculable donc ln est calculable. □

Proposition 44.

$\gamma \in \mathbb{T}$

Démonstration. On sait, par définition que

$$\begin{aligned}
\gamma &= \lim_{n\to+\infty}\left(\sum_{i=1}^{n}\frac{1}{i}-\ln(n)\right)\\
&= \lim_{n\to+\infty}\left(\sum_{i=1}^{n}\frac{1}{i}-\int_1^n\frac{1}{t}\mathrm{d}t\right)\\
&= \lim_{n\to+\infty}\left(\sum_{i=1}^{n-1}\left(\frac{1}{i}-\int_i^{i+1}\frac{1}{t}\mathrm{d}t\right)+\frac{1}{n}\right)\\
&= \lim_{n\to+\infty}\left(\sum_{i=1}^{n-1}\left(\frac{1}{i}-(\ln(i+1)-\ln(i))\right)\right)\\
&= \lim_{n\to+\infty}\left(\sum_{i=1}^{n}\left(\frac{1}{i}-\ln\left(\frac{i+1}{i}\right)\right)\right)\\
&= \lim_{n\to+\infty}\left(\sum_{i=1}^{n}\left(\frac{1}{i}-\ln\left(1+\frac{1}{i}\right)\right)\right)
\end{aligned}$$

Soit $n \in \mathbb{N}^*$,

$$\begin{aligned}
\frac{1}{n}-\left(\ln\left(1+\frac{1}{n}\right)\right) &= \frac{1}{n}-\left(\sum_{i=1}^{+\infty}-\frac{(-\frac{1}{n})^i}{i}\right)\\
&= \frac{1}{n}-\left(\frac{1}{n}+\sum_{i=2}^{+\infty}-\frac{(-\frac{1}{n})^i}{i}\right)\\
&= \sum_{i=2}^{+\infty}-\frac{(-\frac{1}{n})^i}{i}\\
&\leqslant \left|-\frac{(-\frac{1}{n})^2}{2}\right|\\
&\leqslant \frac{1}{2n^2}
\end{aligned}$$

$$
\begin{aligned}
\sum_{i=n}^{+\infty} \frac{1}{2i^2} &= \frac{1}{2} \sum_{i=0}^{+\infty} \frac{1}{(n+i)^2} \\
&= \frac{1}{2} \sum_{i=0}^{+\infty} \frac{1}{n^2 + 2ni + i^2} \\
&\leqslant \frac{1}{2} \sum_{i=1}^{+\infty} \frac{1}{in + i^2} \\
&\leqslant \frac{1}{2} \sum_{i=1}^{+\infty} \frac{1}{i(i+n)} \\
&\leqslant \frac{1}{2} \cdot \frac{\Psi(n+1) + \gamma}{n} \\
&\leqslant \frac{1}{2} \cdot \frac{\frac{\Gamma'(n+1)}{\Gamma(n+1)} + \gamma}{n} \\
&\leqslant \frac{1}{2} \cdot \frac{\frac{n!\left(-\gamma + \sum\limits_{k=1}^{n} \frac{1}{k}\right)}{n!} + \gamma}{n} \\
&\leqslant \frac{1}{2} \cdot \frac{-\gamma + \sum\limits_{k=1}^{n} \frac{1}{k} + \gamma}{n} \\
&\leqslant \frac{1}{2} \cdot \frac{\sum\limits_{k=1}^{n} \frac{1}{k}}{n} \\
&\leqslant \frac{1 + \ln(n)}{2n} \\
&\leqslant \frac{\ln(e) + \ln(n)}{2n} \\
&\leqslant \frac{\ln(en)}{2n}
\end{aligned}
$$

Par conséquent, la précision du calcul augmente avec la quantité $\ln\left(\frac{2n}{\ln(en)}\right)$ et on sait qu'en passant le nombre d'itération au carré à chaque fois, on obtient un bit de bon en plus. □

Proposition 45.

$(\mathbb{T}, +, \times, \cdot)$ est une $\mathbb{Q}$-algèbre à division de dimension dénombrable.

Démonstration. $(\mathbb{T}, +, \times)$ est clairement un corps. En effet, il existe des algorithmes de calcul du produit et de la somme qui sont assez évident à décrire en terme de machine de TURING.

$(\mathbb{T}, +, \cdot)$ est évidemment un $\mathbb{Q}$-espace vectoriel. Il reste à déterminer la dimension de cet espace. $\mathbb{T}$ et $\mathbb{Q}$ étant dénombrables, $\mathbb{T}$ est de dimension au plus dénombrable sur $\mathbb{Q}$.

D'autre part, supposons $\mathbb{T}$ de dimension finie, $\dim(\mathbb{T}) = n$. Le $n+1$-uplet de réels $\left(\pi^0, \ldots, \pi^n\right)$ serait liée. Donc

$$\begin{aligned} \exists (a_0, \ldots, a_n) \in \mathbb{Q}^n : \sum_{i=0}^{n} a_i \pi^i = 0 \\ \exists P \in \mathbb{Q}_n [X] : P(\pi) = 0 \\ \pi \in \mathbb{A} \end{aligned}$$

Or on sait que π est transcendant donc $\mathbb{T}$ ne peut pas être de dimension finie donc $\mathbb{T}$ est de dimension dénombrable sur $\mathbb{Q}$.

□

Il n'est pas nécessaire d'admettre que π est transcendent dans la preuve précédente : l'existence de n'importe quel nombre transcendent suffit. Il y a alors deux méthodes :

— on prouve l'existence d'un nombre transcendent par un argument dénombrement : $\mathbb{A}$ est dénombrable mais $\mathbb{R}$ est en bijection avec $\mathcal{P}(\mathbb{N})$, il existe donc un élément dans $\mathbb{R} \setminus \mathbb{A}$;

— on construit un nombre transcendent : en posant $c = \sum_{n=0}^{+\infty} \frac{1}{10^{n!}}$ on constate que son développement décimal est apériodique puisqu'il ne contient que des 0 et des 1 et que la distance entre deux 1 consécutifs est strictement croissante, en effet

$$\begin{aligned} ((n+1)! - n!)_{n \in \mathbb{N}} &= ((n+1)n! - n!)_{n \in \mathbb{N}} \\ &= (n!(n+1-1))_{n \in \mathbb{N}} \\ &= (n \cdot n!)_{n \in \mathbb{N}} \end{aligned}$$

Proposition 46.

Soit $(f, g) \in \left(\mathbb{R}^{\mathbb{R}}\right)^2$ deux fonctions calculables. La fonction $f \circ g$ est calculable.

Démonstration. En effet, il suffit de composer les machines de TURING qui les calculent. □

On peut étendre la notion de calculabilité aux ordres supérieurs : donner une définition de calculabilité pour les fonctions de fonctions. Par exemple, si f est une fonction calculable, la fonction $x \mapsto f(x+1)$ est aussi calculable. Aussi, il est raisonnable de considérer que la fonction

$$\begin{aligned} &\mathbb{R}^{\mathbb{R}} \to \mathbb{R}^{\mathbb{R}} \\ &\quad f \mapsto (x \mapsto f(x+1)) = f \circ (x \mapsto x+1) \end{aligned}$$

est calculable puisque $(x \mapsto x+1)$ est calculable.

Définition 46.

Soit E et F deux ensembles et $f \in F^E$.
f est dite calculable si il existe une machine de TURING qui pour tout élément t calculable de E associe une de ses descriptions à une description de $f(t)$. Une telle machine est appelée description de f.

On peut ainsi définir des fonctions calculables sur des ensembles de fonctions.

Remarque 7.

On a définit la calculabilité sur des ensembles qu'on peut exprimer par induction :

- les ensembles finis,
- les ensembles $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$ etc.,
- les ensembles de la forme E^n où $n \in \mathbb{N}^*$ et E désigne un ensemble où la calculabilité est définie,
- les ensembles de la forme F^E où on peut définir la calculabilité sur E et sur F.

Le choix des ensembles de base est relativement arbitraire. On peut ajouter tout autre ensemble E pour lequel il existe une injection canonique de cet ensemble dans un ensemble de la forme $A^{(\mathbb{N})}$ où A est un ensemble fini : cela correspond à l'écriture de cet élément avec un alphabet fini. Pour $\mathbb{N}$, il s'agit de l'écriture dans n'importe quelle base, en effet, les chiffres n'ont aucune importance en soit. D'autre part, s'il existe une norme canonique sur E, alors on

peut admettre E s'il existe une injection canonique de cet ensemble dans un ensemble de la forme $A^{\mathbb{N}}$ et tel qu'en écrivant les termes successivement, on ait une convergence vers l'élément voulu.

Plus généralement, si on peut se ramener via une injection canonique à un des ensembles précédemment cité, alors l'ensemble considéré a une bonne structure pour définir des éléments calculables.

Proposition 47.

Pour tout $N \in \mathbb{N}$, $\mathcal{P}^{(n)}(\mathbb{N})$ a une structure de calculabilité.

Démonstration. On fait la preuve par induction.

$n = 0$: $\mathbb{N}$ a une structure de calculabilité.

$n + 1$: soit $n \in \mathbb{N}$. Un élément de $\mathcal{P}^{(n+1)}(\mathbb{N})$ est une partie de $\mathcal{P}^{(n)}(\mathbb{N})$, soit une fonction de $\mathcal{P}^{(n)}(\mathbb{N})$ dans $\{0,1\}$. Or $\mathcal{P}^{(n)}(\mathbb{N})$ a une structure pour la calculabilité d'après l'hypothèse d'induction et $\{0,1\}$ est fini. Donc $\mathcal{P}^{(n+1)}(\mathbb{N})$ a une structure de calculabilité.

D'où la proposition. □

Ainsi, on n'impose pas de limite de cardinal pour qu'un ensemble soit doté d'une structure de calculabilité.

Proposition 48.

Soit E et F des ensembles finis. Toutes les fonctions de E dans F sont calculables.

Démonstration. Les fonctions de E dans F sont en nombre fini. En effet, il y en a $\left|F^E\right| = |F|^{|E|}$.

Il suffit alors d'énumérer toutes les valeurs d'une fonction pour la rendre calculable en prenant une machine de TURING dont l'alphabet contient E et F. □

Proposition 49.

Soit E, F et G trois ensembles. Soit $f \in F^E$ une fonction calculable. Alors la fonction

$$\begin{aligned} G^F &\to G^E \\ g &\mapsto g \circ f \end{aligned}$$

est calculable.

Remarque 8.

Cette proposition n'est intéressant que s'il existe des éléments de G^F qui sont calculables.

Théorème 12.

Deux modèles de calculs définissent le même ensemble de fonctions calculables si, et seulement si, ils définissent le même ensemble de nombres calculables.

Démonstration. $(\Rightarrow)$: On note $\mathfrak{A}$ et $\mathfrak{B}$ deux modèles de calcul. Supposons que les ensembles de fonctions calculées sont égaux. On note respectivement $\mathcal{T}(\mathfrak{A})$ et $\mathcal{T}(\mathfrak{B})$ les ensembles de nombres calculables selon $\mathfrak{A}$ et $\mathfrak{B}$.

Soit $x \in \mathcal{T}(\mathfrak{A})$. On appelle $(a_n)_{n\in\mathbb{N}}$ la suite des décimales de x dans une base b, $b \in \mathbb{N} \setminus \{0,1\}$.

La fonction $f : n \mapsto a_n$ est donc calculable dans le système $\mathfrak{A}$. f est donc calculable dans le système $\mathfrak{B}$. Par conséquent en calculant successivement les valeurs de f pour les entiers, on obtient ainsi un algorithme de calcul des décimales de x. Chaque décimale se calcule en un nombre fini d'étapes puisque cela correspond à l'évaluation de f en un entier ce qui se fait en un nombre fini d'étape.

x est donc calculable selon $\mathfrak{B}$

$(\Leftarrow)$: On note $\mathfrak{A}$ et $\mathfrak{B}$ deux modèles de calcul. Supposons que les ensembles de nombres calculés sont égaux. On note respectivement $\mathcal{F}(\mathfrak{A})$ et $\mathcal{F}(\mathfrak{B})$ les ensembles de fonctions calculables selon $\mathfrak{A}$ et $\mathfrak{B}$.

Soit $f \in \mathcal{F}(\mathfrak{A})$. On constitue la suite $(b_n)_{n\in\mathbb{N}}$ de la façon suivante : on écrit $f(0)$ chiffre par chiffre en binaire puis on place un 2 pour marquer la fin du

nombre ensuite on écrit le nombre $f(1)$...On obtient ainsi une suite décrivant entièrement la fonction. On appelle y le nombre dont la suite des décimales (en base 10) est $(b_n)_{n\in\mathbb{N}}$. Ce nombre y est donc calculable pour le système $\mathfrak{A}$ donc calculable dans le système $\mathfrak{B}$. En décomposant y par la méthode inverse on obtient que la fonction f est calculable selon $\mathfrak{B}$.

D'où l'équivalence annoncée. □

15.5 Le point de vue des ensembles

Définition 47.

Un ensemble est dit récursivement énumérable (ou semi-décidable) s'il est l'image d'une fonction calculable ou l'ensemble vide.

Notation 23.

On note **RE** l'ensemble des langages récursivement énumérables.

Définition 48.

Un ensemble récursif (ou décidable) est un ensemble dont la fonction indicatrice est calculable.

Notation 24.

On note **R** l'ensemble des langages récursifs.

On peut aussi définir un ensemble récursif comme un ensemble dont le test d'appartenance d'un élément à cet ensemble peut être fait grâce à une machine de TURING qui termine toujours.

Le terme décidable s'applique à ce qui peut être résolu par une procédure algorithmique en un nombre fini d'étapes. On comprend alors l'emploi de ce terme dans les définitions ci-dessus.

Proposition 50.

Tout ensemble récursif est récursivement énumérable.

On sait que les machines de TURING sont en nombre dénombrable. On peut donc introduire une bijection ξ entre $\mathbb{N}$ et les machines de TURING. On définit alors l'indice d'une machine de TURING $\mathcal{M}$ comme l'entier n tel que $\xi(n) = \mathcal{M}$. On sait aussi qu'on peut définir une infinité de machines de TURING qui calcule la même fonction. Ainsi une fonction calculable a une infinité d'indice.

Définition 49.

Un ensemble de fonctions est dit extensionnel s'il contient toutes les fonctions partielles de chaque fonction de l'ensemble.
Un ensemble $E \subset \mathbb{N}$ est dit extensionnel quand deux entiers i et j qui sont les indices d'une même fonction calculable vérifie $[i \in E] = [j \in E]$

On peut traduire cela en terme de classe d'équivalence. On introduit la relation d'équivalence $\doteq$ définie sur l'ensemble des machines de TURING par

$$\mathcal{M} \doteq \mathcal{N} :\Leftrightarrow \mathcal{M} \text{ et } \mathcal{N} \text{ calculent la même fonction}$$

On étend cette relations aux entiers :

$$m \doteq n :\Leftrightarrow \xi(m) \text{ et } \xi(n) \text{ calculent la même fonction}$$

Ainsi un sous-ensemble d'entier est extensionnel si il s'agit d'une réunion de classes d'équivalence de $\mathbb{N}/\doteq$. De même un ensemble de machine de TURING est extensionnel si il est une réunion de classes d'équivalence de $\mathbb{M}/\doteq$

On a alors tous les outils pour énoncer un théorème particulièrement général et puissant.

Théorème 13 (RICE).

Tout sous-ensemble extensionnel de $\mathbb{N}$ non trivial (différent de $\mathbb{N}$ et de $\emptyset$) n'est pas récursif.

Démonstration. On montre que la décision du problème de l'arrêt peut se réduire à la décision de toute propriété non triviale sur un ensemble récursivement énumérable. On représente tout ensemble récursivement énumérable par la machine de TURING qui l'accepte. Soit $\mathcal{P}$ une propriété décidable et non triviale sur les ensembles récursivement énumérables. Soit $P(\emptyset) = Faux$ (si ce n'est pas le cas, on raisonne pareillement avec la négation de $\mathcal{P}$). Donc, il existe un ensemble récursivement énumérable $\mathcal{A}$ qui satisfait $\mathcal{P}$. Soit $\mathcal{K}$ une machine de TURING qui accepte $\mathcal{A}$. $\mathcal{P}$ étant décidable, il existe donc une machine de TURING $\mathcal{MP}$ qui, pour toute représentation d'une machine de TURING, décide si l'ensemble récursivement énumérable accepté par cette machine satisfait $\mathcal{P}$ ou non.

Dès lors, on construit une machine de TURING $\mathcal{H}$ prenant en entrée la représentation d'une machine de TURING $\mathcal{M}$ suivie d'une entrée x. L'algorithme de $\mathcal{H}$ est le suivant :

1. Construire la représentation d'une machine de TURING $\mathcal{T}$ qui, sur une entrée d :
   Simule l'exécution de $\mathcal{M}$ sur x
   Simule l'exécution de $\mathcal{K}$ sur d et renvoie le résultat.
2. Simuler l'exécution de $\mathcal{MP}$ sur la représentation de $\mathcal{T}$ et renvoyer le résultat.

Si $\mathcal{M}$ s'arrête sur x, l'ensemble accepté par $\mathcal{T}$ est $\mathcal{A}$. Cet ensemble satisfaisant $\mathcal{P}$, la seconde phase de l'algorithme renverra Vrai. Si $\mathcal{M}$ ne s'arrête pas sur x, l'ensemble accepté par T est $\emptyset$. Cet ensemble ne satisfaisant pas $\mathcal{P}$, la seconde phase de l'algorithme renverra Faux.

La machine $\mathcal{H}$ calcule donc de manière totale le problème de l'arrêt. Celui-ci étant indécidable, par contradiction, $\mathcal{P}$ est donc indécidable. □

On peut donner une interprétation de ce théorème du point de vu des ensembles récursivement énumérables.

Corollaire 12.

Toute propriété non triviale (qui n'est pas constamment vraie ou fausse) sur les ensembles récursivement énumérables est indécidable.

Le problème de l'arrêt est en fait une conséquence du théorème de RICE. Il s'avère même que beaucoup de fonctions incalculables sont des conséquences du théorème de RICE.

Chapitre 16

Les machines de TURING à oracles

16.1 Définition

Les machines de TURING calculent une classe de fonction restreinte : les fonctions semi-calculables. Cependant, on réalise que des calculs peuvent s'effectuer si on autorise des étapes qui consistent en l'exécution d'opérations incalculables.

Ainsi on définit un type de machines de TURING qui peut réaliser en une étape des fonctions incalculables.

Définition 50 (Machine de TURING à oracle pour la décision)**.**

Une machine de TURING à oracle pour la décision est un 11-uplet $(Q, \Gamma, B, \Sigma, q_0, \delta, F, q_?, q_Y, q_N, A)$ où :

- Q est un ensemble fini d'états,
- Γ est l'alphabet de travail,
- $B \in \Gamma$ est un symbole particulier dit symbole blanc,
- Σ est l'alphabet des symboles en entrée ($\Sigma \subseteq \Gamma \setminus \{B\}$)
- $q_0 \in Q$ est l'état initial,
- $\delta : Q \setminus \{q_?\} \times \Gamma \to Q \times \Gamma^2 \times \left\{ \{\longleftarrow, \longrightarrow\}^2 \times \{\texttt{HALT}\} \right\}$ est la fonction de transition,
- $F \subseteq Q$ est l'ensemble des états acceptants.
- $(q_?, q_Y, q_N) \in Q^3$ et $q_? \notin \{q_Y, q_N\}$
- $A \subseteq \Gamma^\star$

Il s'agit d'une machine de TURING avec oracle sur A.

Une machine de TURING à oracle est une machine de TURING comportant un ruban particulier appelé ruban d'oracle. Lors de l'exécution de la machine

de TURING, la machine peut passer dans l'état $q_?$. À ce moment, la machine effectue plusieurs opérations en une seule étape à la place d'une transition. La machine passe dans l'état q_Y si le mot sur le ruban d'oracle appartient à A et dans q_N sinon et le mot du ruban d'oracle est effacé. Après, la machine reprend son exécution.

On donne une autre définition des machines de TURING à oracle faite pour les problèmes de calcul.

Définition 51 (Machine de TURING à oracle pour le calcul)**.**

Une machine de TURING (pour le calcul) à oracle est un 10-uplet $(Q, \Gamma, B, \Sigma, q_0, \delta, F, q_?, q_!, f)$ où :

- Q est un ensemble fini d'états,
- Γ est l'alphabet de travail,
- $B \in \Gamma$ est un symbole particulier dit symbole blanc,
- Σ est l'alphabet des symboles en entrée ($\Sigma \subseteq \Gamma \setminus \{B\}$)
- $q_0 \in Q$ est l'état initial,
- $\delta : Q \setminus \{q_?\} \times \Gamma \to Q \times \Gamma^2 \times \left\{\{\longleftarrow, \longrightarrow\}^2 \times \{\texttt{HALT}\}\right\}$ est la fonction de transition,
- $F \subseteq Q$ est l'ensemble des états acceptants.
- $(q_?, q_!) \in Q^2$ et $q_? \neq q_!$
- $f \in (\Gamma^\star)^{(\Gamma^\star)}$

Il s'agit d'une machine de TURING avec oracle sur f.

Cette fois, lors du passage dans $q_?$, le mot w du ruban d'oracle est effacé et le mot $f(w)$ est inscrit à la place en plaçant la première lettre sur la tête de lecture de ce ruban, ensuite la machine passe dans l'état $q_!$. Ce modèle est clairement plus orienté vers le calcul que la décision mais reste d'usage rarissime.

16.2 Propriétés

Proposition 51.

Soit A un langage décidable. Toute machine de TURING à oracle sur A reconnait exactement les langages décidables.

Proposition 52.

Soit f une fonction calculable. Toute machine de TURING à oracle sur f calcule exactement les fonctions semi-calculables.

Chapitre 17

Comparaison de problèmes

17.1 Réduction de TURING

Une méthode usuelle pour comparer des problèmes A et B en calculabilité consiste à déterminer si avoir la solution de A fourni une solution à B. Alors, on peut intuitivement dire que B est moins difficile que A. Si, en plus, une solution à B permet de trouver une solution à A, on peut dire que ces problèmes sont également difficiles.

Cette méthode est la réduction de TURING. Formellement, on ne s'intéresse qu'aux problèmes de décision. On peut aussi donner une définition de réduction pour les problèmes de calcul, l'équivalence est simple et ne sera pas montrée ici. Du point de vue d'une machine de TURING, une solution pour le problème A est un oracle sur A.

De plus on restreint aussi les problèmes de décisions considérés : on ne s'intéresse qu'à la décision de l'appartenance d'un naturel à une partie de $\mathbb{N}$. Il est évident que se limiter à cette classe de problèmes de décisions ne réduit pas la puissance ou la complexité des tâches.

Le terme d'ensemble utilisé sans précision désignera dans cette section un sous ensemble de $\mathbb{N}$.

Définition 52 (Réduction de TURING)**.**

Étant deux ensembles $(A, B) \in (\mathcal{P}(\mathbb{N}))^2$, on dit que A est TURING-réductible (ou réductible au sens de TURING) à B s'il existe une machine de TURING à oracle sur le problème de l'appartenance à B qui décide le problème de l'appartenance à A.

On peut aussi voir ça en terme de calcul de la fonction indicatrice de A avec une machine qui a la fonction indicatrice de B pour oracle.

L'information de l'appartenance à B permet à elle seule à décider l'appartenance à A. Autrement dit, A est plus facile que B. Cette interprétation justifie la notation suivante.

Notation 25.

Si A est TURING-réductible à B, on écrit

$$A \leqslant_T B$$

et on dit que A est B-récursivement énumérable.

En particulier, les ensembles calculables sont donc les plus faciles de tous puisque, par définition, on peut les décider sans oracle. L'ajout d'un oracle est donc inutile. Réciproquement, une machine avec un oracle sur un ensemble calculable n'est pas plus puissante qu'une machine sans oracle.

Définition 53.

On dit que deux ensembles A et B sont TURING-équivalent (équivalent au sens de TURING) si $A \leqslant_T B$ et $B \leqslant_T A$.

Il ne faut pas confondre les différents sens donnés à l'expression TURING-équivalent. À propos d'un modèle de calcul, cela désigne une puissance de calcul similaire à celle des machines de TURING. Au sujet d'un problème de décision, cela signifie que pour une machine de TURING, il est également difficile de décider l'un et l'autre.

Notation 26.

Si A et B sont TURING-équivalent, on écrit

$$A \equiv_T B$$

Proposition 53.

$\equiv_T$ est une relation d'équivalence sur $\mathcal{P}(\mathbb{N})$.

Démonstration. La réflexivité est immédiate.

La symétrie et la transitivité provient directement de la définition. □

17.2 Degrés de TURING

Étant donné que $\equiv_T$ est une relation d'équivalence sur les problèmes de décision, on peut aisément faire des classes de problèmes également difficiles. Ces classes sont appelées degré de TURING.

Définition 54 (Degré de TURING)**.**

On appelle degré de TURING toute classe d'équivalence de $\mathcal{P}(\mathbb{N}) / \equiv_T$.

Un degré de TURING correspond donc à une classe de problèmes également difficiles à décider pour un modèle de calcul TURING-équivalent.

Notation 27.

Le degré de TURING d'un ensemble E est noté $\deg(E)$ et correspond au degré de TURING auquel E appartient.

Notation 28.

On note $\emptyset$ le degré de TURING des ensembles décidable.

En effet, ce degré correspond au degré qui contient l'ensemble vide (qui est décidable). En effet, pour une machine de TURING, les problèmes décidables sont également difficile à décider : cela peut se faire sans oracle, donc à fortiori avec un oracle quelconque qu'on ignore.

Définition 55.

Étant donné $\mathcal{B} \subseteq \mathcal{P}(\mathbb{N})$, $A \in \mathcal{P}(\mathbb{N})$ est dit TURING-difficile pour $\mathcal{B}$ si

$$\forall B \in \mathcal{B}, B \leqslant_T A$$

Si, en plus, $A \in \mathcal{B}$, alors A est dit TURING-complet pour $\mathcal{B}$.

La TURING-difficulté signifie que A est plus difficile que n'importe quel problème de $\mathcal{B}$. La TURING-complétude signifie que A est un des problèmes les plus durs parmi $\mathcal{B}$.

Proposition 54.

$\leqslant_T$ est transitif.

Corollaire 13.

$\leqslant_T$ est un préordre sur $\mathcal{P}(\mathbb{N})$

De plus, on n'a pas l'antisymétrie, on sait donc que ce n'est pas un ordre. En effet, l'antisymétrie impliquerait que deux problèmes également difficiles sont identiques. Ce qui est évidemment faux. Par exemple, pour tout $X \in \mathcal{P}(\mathbb{N}) \setminus \{\mathbb{N}, \emptyset\}$, X et $\{x+1 \mid x \in X\}$ sont évidemment également difficiles mais différents.

Par ailleurs, comme toujours, on peut déduire une relation d'équivalence d'un préordre. Dans ce cas, c'est la relation $\equiv_T$ déjà vu. En quotientant $\mathcal{P}(\mathbb{N})$ par cette relation, on obtient l'ensemble des degrés de TURING sur lequel $\leqslant_T$ est une relation d'ordre.

Proposition 55.

$$\forall A \in \mathcal{P}(\mathbb{N}), A \equiv_T \mathbb{N} \setminus A$$

Démonstration. Pour décider si $x \in \mathbb{N}$ est un élément de $\mathbb{N} \setminus A$, il suffit de décider $x \in A$ et d'inverser le résultat. La réciproque se traite de la même façon. □

Proposition 56.

Tout ensemble décidable est TURING-réductible à n'importe quel ensemble décidable.

Démonstration. Un ensemble est décidable si on peut décider l'appartenance d'un élément à cet ensemble avec une machine de TURING sans oracle. En ajoutant un oracle, on ne diminue pas la puissance puisqu'on peut ignorer l'oracle. Aussi la réduction de tout ensemble décidable à tout ensemble décidable est vraie. □

Corollaire 14.

Tout problème décidable est complet pour la classe des problèmes décidables.

On donne plusieurs propositions sur les degrés de TURING qu'on ne montrera pas. Elle ne sont là que pour faire intuiter la structure de ces ensembles.

Proposition 57.

Chaque degré de TURING contient un nombre dénombrable d'ensemble.

Démonstration. Soit un degré de TURING X et un problème A contenu dans X. X est donc l'ensemble des problèmes décidés par une machine de TURING avec A comme oracle. Or, il n'existe qu'un nombre dénombrable de machines de TURING pour n'importe quel oracle donné. Par conséquent tout degré de TURING est dénombrable. □

Proposition 58.

Les degrés de TURING sont en bijection avec $\mathbb{R}$.

Définition 56.

Un degré a est minimal si a n'est pas $\emptyset$ et qu'il n'y a pas de degré entre $\emptyset$ et a.

Proposition 59.

Il existe des degrés minimaux.

Corollaire 15.

La relation d'ordre sur les degrés n'est pas un ordre dense.

Proposition 60.

Pour tout degré de TURING a différent de $\emptyset$, il existe un autre degré incomparable avec a.

Proposition 61.

L'ensemble des couples de degrés incomparables en bijection avec $\mathbb{R}$.

17.3 Saut de TURING

On a vu précédemment que les machines de TURING sont incapables de décider l'arrêt. Plus généralement, toute machine de Turing avec oracle est incapable de décider son propre arrêt ou celui de toute machine munie d'un oracle du même degré.

Cependant, on peut doter une autre machine d'un oracle plus puissant afin de déterminer l'arrêt de la précédente. Ainsi, à chaque degré de TURING, on associe un degré supérieur qui permet de décider l'arrêt des machines à oracle sur ce premier degré. Cette opération est appelé un saut de TURING. Elle permet d'atteindre un degré strictement supérieur, cependant, l'écart entre ces degré est non vide.

Dans toute cette section on se donne une numérotation indexée par les entiers naturels des machines de TURING à oracle pour la décision.

Définition 57.

Soit X un degré de TURING. On défini le saut de TURING de X comme le degré de l'ensemble des indices des machines de TURING à oracle sur X qui s'arrête sur le ruban vide.

Notation 29.

On note X' le saut de TURING de X.
Récursivement, on note

$$X^{(n)} = \begin{cases} X & \text{si } n = 0 \\ \left(X^{(n-1)}\right)' & \text{sinon} \end{cases}$$

Proposition 62.

$\emptyset'$ est équivalent au problème de l'arrêt.

Proposition 63.

$$\forall (A, B) \in \mathcal{P}\left(\mathbb{N}\right)^2, A \equiv_T B \Rightarrow A' \equiv_T B'$$

Proposition 64.

Pour tout degré a, il y a un degré b tel que $a <_T b$ et $a' \equiv_T b'$.

Proposition 65.

$$\forall A \in \mathcal{P}(\mathbb{N}), A \leqslant_T A'$$

Proposition 66.

Pour tout degré A, il y a un degré se trouvant strictement entre A et A'.

Proposition 67.

Un degré a est un saut si $\emptyset' \leqslant_T a$.

Proposition 68.

Il y a une suite $(a_i)_{i\in\mathbb{N}}$ de degrés telle que

$$\forall i \in \mathbb{N}, a'_{i+1} \leqslant_T a_i$$

Chapitre 18

La hiérarchie de GRZEGORCZYK

18.1 Les fonctions élémentaires

Ce chapitre décrit une hiérarchie qui ordonne finement la classe des fonctions récursives primitives en une infinité dénombrable de classes. Ce chapitre est largement inspiré de la publication de GRZEGORCZYK fondant cette hiérarchie [Grz53] et, dans une moindre mesure, d'un livre de H.E.ROSE traitant du même sujet [Ros84].

Cette hiérarchie compare la croissance des fonctions primitives récursives. On associe à chaque classe, les croissances des itérées des fonctions de la classe précédente. On capture ainsi toutes les fonctions primitives récursives. Toutefois, on ne capture pas les fonctions récursives qui ont une croissance qui peut être arbitrairement plus importante.

Ces classes sont définies par induction. On donne des fonctions de bases et des schémas de constructions rappelant les schémas de constructions primitifs. En effet, la construction récursive permet d'atteindre des croissances non bornées. Il est nécessaire de modifier cette construction pour borner la croissance des fonctions ainsi engendrées. En outre le schéma de composition est légèrement adapté mais ne change pas fondamentalement.

Notation 30.

$$\begin{aligned} \dot{-} : \mathbb{N}^2 &\twoheadrightarrow \mathbb{N} \\ (a, b) &\mapsto \max(0, a - b) \end{aligned}$$

utilisé en notation infixe.

Définition 58 (Les opérations de substitution).

On distingue dans les opérations de substitution trois types d'opération. Une classe de fonctions X est dite stable pas substitution si :

— étant donné $(f, g) \in X\left(\mathbb{N}^n, \mathbb{N}^p\right) \times X\left(\mathbb{N}^m, \mathbb{N}\right)$, alors

$$\begin{aligned}&\forall k \in [\![1, n]\!], ((x_1, \ldots, x_n, y_1, \ldots, y_m) \mapsto \\ &f(x_1, \ldots, x_{k-1}, g\left(y_1, \ldots, y_n\right), x_{k+1}, \ldots, x_n)) \in X\end{aligned}$$

— étant donné $f \in X\left(\mathbb{N}^n, \mathbb{N}^m\right)$ et $(j, k) \in [\![1, n]\!]^2$ avec $j < k$, alors

$$\begin{aligned}&\left(x_1, \ldots, x_{j-1}, y, x_{k+1}, \ldots x_n\right) \mapsto \\ &f\left(x_1, \ldots, x_{j-1}, y, \ldots, y, x_{k+1}, \ldots x_n\right) \in X\end{aligned}$$

— étant donné $f \in X\left(\mathbb{N}^n, \mathbb{N}^m\right)$ et $j \in [\![1, n]\!]$, alors

$$\begin{aligned}&\left(x_1, \ldots, x_{j-1}, x_{j+1}, \ldots, x_n\right) \mapsto \\ &f\left(x_1, \ldots, x_{j-1}, 0, x_{j+1}, \ldots, x_n\right) \in X\end{aligned}$$

Définition 59 (Fonction élémentaire).

La classe des fonctions élémentaires est le plus petit ensemble de fonction contenant $x \mapsto x + 1$, $(x, y) \mapsto x + y$ et $(x, y) \mapsto x \dot{-} y$ et stable par les opération de substitution et par somme bornée (SumB $()$) et produit borné (ProdB $()$).

Notation 31.

La classe des fonctions élémentaires est noté $\mathcal{E}$. On trouve aussi l'appellation *ELEMENTARY* dans la littérature.

Exemple 21.

$(x, y) \mapsto xy$, $(x, y) \mapsto x^y$ et $x \mapsto x!$ sont des fonctions élémentaires.

Proposition 69.

$\mathcal{E}$ est stable par minimisation bornée.

Démonstration. Soit $f \in \mathcal{E}(\mathbb{N}^n \times \mathbb{N}, \mathbb{N}^m)$.

$$h(u,y) = \sum_{i=0}^{y} (1 \dot{-} f(u,i))$$

et

$$g(u,x) = \left(\sum_{y=0}^{x} (1 \dot{-} f(u,y))\right)\left(1 \dot{-} \left(\left(\sum_{y=0}^{x} (1 \dot{-} f(u,y))\right) \dot{-} x\right)\right)$$

On a $g(u,x) = \text{MinB}\,(x \mapsto f(u,x))(x)$.

□

Corollaire 16.

Pour toute classe X de fonctions stable pour les opérations de substitution, de somme bornée, par $(x,y) \mapsto x \dot{-} y$ et $(x,y) \mapsto x(1 \dot{-} y)$, X est stable par minimisation bornée.

Démonstration. Reprendre la preuve précédente. □

Définition 60.

Un prédicat est dit élémentaire si sa fonction indicatrice est élémentaire.

Proposition 70.

La classe des prédicats élémentaire est stable par quantification bornée.

Démonstration. On reprend la preuve pour les prédicats primitifs récursifs qui n'utilise que des fonctions élémentaires. □

Corollaire 17.

Pour toute classe X stable pour les opération de minimisation bornée et de substitutions, l'ensemble des prédicats dont la fonction indicatrice est dans X est stable pour les opérations des quantificateurs bornés.

Remarque 9.

Les prédicats élémentaires sont stables pour les opérations logiques (comme $\vee$, $\wedge$, $\neg$...).

On redéfinit un certain nombre de fonctions de bases avec ce formalisme. Le but étant de trouver des résultats sur les fonctions élémentaires. On remarque qu'on n'interprète rien comme des prédicats mais comme des fonctions à valeur dans $\{0,1\}$: leur fonction caractéristique.

— $((x,y) \mapsto [x \leqslant y]) := (x,y) \mapsto [x \dot{-} y = 0]$
— $((x,y) \mapsto [x = y]) := (x,y) \mapsto (x \leqslant y)\,(y \leqslant x)$
— $((x,y) \mapsto [x < y]) := (x,y) \mapsto (x \leqslant y)\,(1 \dot{-} (x = y))$
— $((x,y) \mapsto [x|y]) := (x,y) \mapsto \sum_{z \in [\![0,y]\!]} (xy = z)$
— $(x \mapsto prime(x)) := x \mapsto \left(\prod_{y \in [\![0,x]\!]} y|x \Rightarrow (y = 1 \vee y = x) \right)(x \geqslant 2)$
— $\left((x,y) \mapsto \left\lfloor \frac{x}{y} \right\rfloor\right) := (x,y) \mapsto MinB\,(z \mapsto (x+1 \leqslant y(z+1)))\,(x)$
— $((x,y) \mapsto r(x,y)) := (x,y) \mapsto x \dot{-} \left(y \left\lfloor \frac{x}{y} \right\rfloor\right)$
— $(x \mapsto x \uparrow\uparrow 2) := x \mapsto x^x$
— $(x \mapsto x \uparrow\uparrow 3) := x \mapsto x^{x \uparrow\uparrow 2}$
— $\left((x,y) \mapsto \left\lfloor \sqrt[y]{x} \right\rfloor\right) :=$

$$(x,y) \mapsto MinB \left(z \mapsto [z^y \leqslant x] \prod_{t \in [\![0,x]\!]} [t^y \leqslant x \Rightarrow t \leqslant z] \right)(x)$$

Dans la suite, on utilise deux opérations de maximisation :

$$\max_{x \in [\![0,z]\!]} \{x \mid R(u,x)\} = \text{MinB} \left(x \mapsto R(u,x) \left(\prod_{t \in [\![0,z]\!]} [R(u,t) \Rightarrow t \leqslant x] \right) \right)(z)$$

(le plus grand $x \leqslant z$ tel que $R(u,x)$) et

$$\underset{x\in[\![0,z]\!]}{\mathrm{Max}}\, F(u,z) = F\left(u, \mathrm{MinB}\left(x \mapsto \left(\prod_{t\in[\![0,z]\!]} [F(u,t) \leqslant F(u,x)]\right)\right)(z)\right)$$

(la valeur maximale de $F(u,x)$ pour $x \leqslant z$).

Remarque 10.

$\mathcal{E}$ est stable par ces deux opérations de maximisation.

18.1.1 Les nombres premiers

On se donne d'autres opérations arithmétiques.

$$((x,y) \mapsto \exp(y,k)) := (x,y) \mapsto \max_{i\in[\![0,y]\!]} \left\{ i \;\middle|\; y \equiv 0 \left[k^i\right] \right\}$$

(le plus grand exposant i pour lequel y est divisible par k^i)

$$((x,y) \mapsto [x\, N\, y]) := (x,y) \mapsto \mathit{prime}(x)\, \mathit{prime}(y)\, [x > y] \left(\prod_{z\in[\![0,x]\!]} [y < z \wedge \mathit{prime}(z) \Rightarrow z = x]\right)$$

(x et y sont deux nombres premiers consécutifs)

$$(x,k) \mapsto \sum_{v\in[\![0,(k+2)\uparrow\uparrow 3]\!]} \left[[\exp(v,2) = 2]\, \mathit{prime}(x) \exp(v,x) = k \right.$$
$$+ 2 \cdot \prod_{t\in[\![0,(k+2)\uparrow\uparrow 2]\!]} \prod_{z>t} \left[\mathit{prime}(z)\ \mathit{prime}(t)\ [z|v]\ [t|v] = 1 \Rightarrow \right.$$
$$\left.\left. \sum_{w\in[\![0,t]\!]} ([w|v]\ [w\, N\, z]\ [\exp(v,w) = exp(v,z) + 1]) \right] \right]$$

(x est le $k^{\text{ème}}$ nombre premier)

$$p_k = \mathrm{MinB}\,(x \mapsto x\ \mathrm{Pr}\ k)((k+2) \uparrow\uparrow 2)$$

(le $k^{\text{ème}}$ nombre premier).

Remarque 11.

On remarque que $\forall k \in \mathbb{N}, p_k \leqslant (k+2)^{k+2} = (k+2) \uparrow\uparrow 2$, ce qui justifie la définition précédente.

18.1.2 Représentation des n-uplets par des entiers

Grâce aux fonctions précédentes, on peut représenter tout n-uplet par un entier.

Prenons le n-uplet $(m_0, \ldots, m_{n-1}) \in \mathbb{N}^n$, on associe le nombre

$$m = \prod_{i=0}^{n-1} p_i^{m_i}$$

On a alors $\forall i \in [\![0, n-1]\!], m_i = \exp(m, p_i)$.

18.1.3 La récursion bornée

Définition 61 (Construction récursive bornée de fonction)**.**

Soit $(k, r) \in \mathbb{N}^2$, f une fonction de $\mathbb{N}^k$ dans $\mathbb{N}^r$, g une fonction de $\mathbb{N}^{k+r+1}$ dans $\mathbb{N}^r$ et j une fonction de $\mathbb{N}^{k+1}$ dans $\mathbb{N}^r$. La fonction $h = \mathrm{RecB}(f, g, j)$ est la fonction de $\mathbb{N}^{k+1}$ dans $\mathbb{N}^r$ définie pour tout $n \in \mathbb{N}$ et pour tout $m \in \mathbb{N}^k$ par :

$$\begin{aligned} h(0, m) &= f(m) \\ h(n+1, m) &= g(n, h(n, m), m) \\ h(n, m) &\leqslant j(n, m) \end{aligned}$$

Exemple 22.

Pour toute classe X de fonction stable par les opérations de substitution, de récursion bornée et qui contient les fonctions $x \mapsto x+1$ et $(x, y) \mapsto x^y$ alors X contient aussi $(x, y) \mapsto x + y$ et $(x, y) \mapsto xy$.

Démonstration. Il suffit en effet de définir la somme et le produit comme précédemment en les bornant par la fonction puissance. □

Proposition 71.

$\mathcal{E}$ est stable par récursion bornée.

Démonstration. Soit des fonctions telles que $f = \text{RecB}\,(g, h, j)$.

Si $y = f(u, x)$, il existe une séquence finie $(m_0, \ldots, m_x) \in \mathbb{N}^{x+1}$ telle que

$$\begin{aligned} m_0 &= g(u) \\ m_1 &= h(u, 0, m_0) \\ &\vdots \\ m_x &= h(u, x-1, m_{x-1}) \\ y &= m_x \end{aligned}$$

et ce $x+1$-uplet est unique.

Soit $m = \prod_{i=0}^{x} p_i^{m_i}$ et ainsi $\forall k \in [\![0, x-1]\!], [\exp(m, p_{k+1} = h(u, k, \exp(m, p_k)])$

On pose

$$F(x,u) = p_x^{(x+2)\underset{z\in[\![0,x]\!]}{\text{Max}} f(u,z)}$$

$$(y = f(u,x)) := \sum_{m\in[\![0,F(x,u)]\!]} \left[[\exp(m,2) = g(u)] \left[y = \exp(m, p_x) \right. \right.$$

$$\left. \left. \prod_{k\in[\![0,x-1]\!]} [\exp(m, p_{k+1}) = h(u, k, \exp(m, p_k))] \right] \right]$$

Comme p_x est le plus grand facteur premier de m et $m_i \leqslant j(u,i)$, on a $\exp(m, p_i) \leqslant \underset{i\in[\![0,x]\!]}{\text{Max}}\, j(u,i)$. Prenons $R(y,u,x)$ pour la dernière relation. La relation R est élémentaire si les fonctions g, h et j le sont. D'après la dernière équation, il vient que la fonction f peut être définie grâce à la minimisation bornée de la façon suivante

$$f(u,x) = MinB\,(y \mapsto R(y,u,x))\,(j)$$

Donc f est élémentaire. □

18.1.4 Définitions équivalentes

Définition 62.

On note $\mathcal{E}'$ le plus petit ensemble de fonction contenant $x \mapsto x+1$, $(x,y) \mapsto x \dot{-} y$ et $(x,y) \mapsto x^y$ et stable par les opération de substitution et par minimisation bornée.

Théorème 14.

$$\mathcal{E} = \mathcal{E}'$$

Démonstration. ($\mathcal{E}' \subseteq \mathcal{E}$) : Cette inclusion se fait en montrant que $(x,y) \mapsto x^y$ appartient à $\mathcal{E}$ et que cette classe est stable par minimisation bornée (cf 69).

($\mathcal{E} \subseteq \mathcal{E}'$) : Pour cette inclusion, il faut prouver que $(x,y) \mapsto x+y$ appartient à $\mathcal{E}'$ et que $\mathcal{E}'$ est stable par somme bornée et produit borné.

Notons tout d'abord que $\mathcal{E}'$ est stable pour les opération du calcul propositionnel et par les quantificateurs bornés.

De plus les relation $\leqslant$, $\geqslant$ et $=$ sont aussi des relations de $\mathcal{E}'$. Donc on voit facilement que $\mathcal{E}'$ est stable par les deux opérations de maximisation. Par conséquent les fonctions

$$\begin{aligned} xy &= MinB(z \mapsto [(2^x)^y = 2^z])((x+1)^{y+1}) \\ x+y &= MinB(z \mapsto [(2^x)2^y = 2^z])((x+1)(y+1)) \end{aligned}$$

sont dans $\mathcal{E}'$.

D'autre part, $x \mapsto x \uparrow\uparrow 2$, $x \mapsto x \uparrow\uparrow 3$, $x \mapsto prime(x)$, $(x,y) \mapsto \exp(x,y)$, $(x,y) \mapsto x\, N\, y$ et $x \mapsto p_x$ sont des fonctions de $\mathcal{E}'$. Donc $\mathcal{E}'$ est stable par récursion bornée.

On peut maintenant facilement prouver que $\mathcal{E}'$ est stable par somme bornée et produit borné. On se donne $F \in \mathcal{E}'$ et on pose

$$\begin{aligned} f(u,x) &= \sum_{i \in [\![0,x]\!]} F(u,i) \\ g(u,x) &= \prod_{i \in [\![0,x]\!]} F(u,i) \end{aligned}$$

On a

$$
\begin{aligned}
f(u,0) &= F(u,0)\\
f(u,x+1) &= f(u,x) + F(u,x+1)\\
f(u,x) &\leqslant (x+1) \operatorname*{Max}_{z\in[\![0,x]\!]} F(u,z)\\
g(u,0) &= F(u,0)\\
g(u,x+1) &= f(u,x)F(u,x+1)\\
g(u,x) &\leqslant \left(\operatorname*{Max}_{z\in[\![0,x]\!]} F(u,z)\right)^{x+1}
\end{aligned}
$$

Donc les fonctions f et g appartiennent aussi à la classe $\mathcal{E}'$. Par conséquent $\mathcal{E}'$ est stable par toutes les opérations de $\mathcal{E}$ donc $\mathcal{E} \subseteq \mathcal{E}'$ □

Définition 63.

On note $\mathcal{E}''$ le plus petit ensemble de fonction contenant $x \mapsto x+1$ et $(x,y) \mapsto x^y$ et stable par les opération de substitution et par récursion bornée.

Théorème 15.

$$\mathcal{E} = \mathcal{E}''$$

Démonstration. ($\mathcal{E}'' \subseteq \mathcal{E}$) : Découle directement de 71.

($\mathcal{E} \subseteq \mathcal{E}''$) : On prouve d'abord $\mathcal{E}' \subseteq \mathcal{E}''$ et on utilise le théorème 14.

Tout d'abord, on peut facilement montré que les fonctions initiales de $\mathcal{E}'$ contient $(x,y) \mapsto x \dot{-} y$, $x \mapsto x \dot{-} 1$, $(x,y) \mapsto x+y$ et $(x,y) \mapsto xy$ appartiennent à $\mathcal{E}''$.

Maintenant, on peut prouver que la classe $\mathcal{E}''$ est stable par une version faible de la somme bornée : avec $F \in \mathcal{E}''$

$$f(u,x) = \sum_{i\in[\![0,x]\!]} (1 \dot{-} F(u,i))$$

$f \in \mathcal{E}''$ puisqu'elle peut être définie par

$$
\begin{aligned}
f(u,0) &= 1 \dot{-} F(u,0)\\
f(u,x+1) &= f(u,x) + (1 \dot{-} F(u,x+1))\\
f(u,x) &\leqslant x+1
\end{aligned}
$$

Il vient que $\mathcal{E}''$ est stable par minimisation bornée. Par conséquent $\mathcal{E}' \subseteq \mathcal{E}''$ et donc $\mathcal{E} = \mathcal{E}''$ □

Définition 64.

On note $\mathcal{E}'''$ le plus petit ensemble de fonction contenant $x \mapsto x+1$, $(x,y) \mapsto x \dot{-} y$, $(x,y) \mapsto xy$ et $(x,y) \mapsto x^y$ et stable par les opération de substitution et par somme bornée.

Théorème 16.

$$\mathcal{E} = \mathcal{E}'''$$

Démonstration. $(\mathcal{E}''' \subseteq \mathcal{E})$: par définition

$(\mathcal{E} \subseteq \mathcal{E}''')$: $\mathcal{E}'''$ est clos par minimisation bornée. □

18.2 Les classes $\mathcal{E}^n$

On définit la suite de fonction de $\mathbb{N}^2$ dans $\mathbb{N}$ suivante

$$\begin{aligned}
\mathfrak{f}_0 : (x,y) &\mapsto y+1 \\
\mathfrak{f}_1 : (x,y) &\mapsto x+y \\
\mathfrak{f}_2 : (x,y) &\mapsto (x+1)(y+1) \\
\forall n \in \mathbb{N}\setminus\{0,1\}, \forall y \in \mathbb{N}, \mathfrak{f}_{n+1}(0,y) &= \mathfrak{f}_n(y+1,y+1) \\
\forall n \in \mathbb{N}\setminus\{0,1\}, \forall (x,y) \in \mathbb{N}^2, \mathfrak{f}_{n+1}(x+1,y) &= \mathfrak{f}_{n+1}(x,\mathfrak{f}_{n+1}(x,y))
\end{aligned}$$

Proposition 72.

$$\forall n \in \mathbb{N}\setminus\{0,1\}, \forall (x,y) \in \mathbb{N}^2, \mathfrak{f}_n(x,y) > y$$

Démonstration. Preuve par récurrence sur n.

(Initialisation) : Si $n = 2$, on a $\mathfrak{f}_2(x,y) = (x+1)(y+1) > y$.

(Hérédité) : Supposons maintenant que le théorème est vérifié au rang $n \geqslant 2$. De l'hypothèse de récurrence, il vient

$$\mathfrak{f}_{n+1}(0,y) = \mathfrak{f}_n(y+1,y+1) > y+1 > y$$

Supposons que l'inégalité

$$\mathfrak{f}_{n+1}(x,y) > y$$

est vraie pour un x donné et pour tout y. Donc elle est aussi vrai pour $x+1$

$$\begin{aligned}
\mathfrak{f}_{n+1}(x+1,y) &= \mathfrak{f}_{n+1}(x,\mathfrak{f}_{n+1}(x,y)) \\
&> \mathfrak{f}_{n+1}(x,y) \\
&> y
\end{aligned}$$

Donc le théorème est aussi vrai pour tout x et n. □

Proposition 73 (Croissance stricte par rapport à la première variable).

$$\forall n \in \mathbb{N}, \forall (x,y) \in \mathbb{N}^2, \mathfrak{f}_{n+1}(x+1,y) > \mathfrak{f}_{n+1}(x,y)$$

Démonstration. Pour $n \geqslant 2$, grâce au théorème précédent, on a

$$\begin{aligned} \mathfrak{f}_{n+1}(x+1,y) &= \mathfrak{f}_{n+1}(x,\mathfrak{f}_{n+1}(x,y)) \\ &> \mathfrak{f}_{n+1}(x,y) \end{aligned}$$

Pour $n \in \{0,1\}$, on vérifie directement

$$x+1+y > x+y$$

et

$$(x+2)(y+1) > (x+1)(y+1)$$

□

Proposition 74 (Croissance stricte par rapport à la seconde variable)**.**

$$\forall n \in \mathbb{N}^*, \forall (x,y) \in \mathbb{N}^2, \mathfrak{f}_n(x,y+1) > \mathfrak{f}_n(x,y)$$

Démonstration. Pour $n \in \{1,2\}$, on vérifie le théorème directement. Pour $n \geqslant 2$, on fait la preuve par récurrence.

Supposons que l'inégalité est vraie pour un n donné et pour tout x et y. Grâce au théorème précédent, on a

$$\begin{aligned} \mathfrak{f}_{n+1}(0,y+1) &= \mathfrak{f}_n(y+2,y+2) \\ &> \mathfrak{f}_n(y+1,y+2) \\ &> \mathfrak{f}_n(y+1,y+1) = \mathfrak{f}_{n+1}(0,y) \end{aligned}$$

Supposons la propriété vraie au rang $n+1$ pour un x donné et pour tout y. On trouve qu'elle est aussi aussi vraie pour $x+1$, en effet

$$\begin{aligned} \mathfrak{f}_{n+1}(x+1,y+1) &= \mathfrak{f}_{n+1}(x,\mathfrak{f}_{n+1}(x,y+1)) \\ &> \mathfrak{f}_{n+1}(x,\mathfrak{f}_{n+1}(x,y)) = \mathfrak{f}_{n+1}(x+1,y) \end{aligned}$$

□

Définition 65.

Pour tout $n \in \mathbb{N}$, on note $\mathcal{E}_n$ le plus petit ensemble de fonction contenant $x \mapsto x+1$, $\pi_1 : (x,y) \mapsto x$, $\pi_2 : (x,y) \mapsto y$ et $(x,y) \mapsto \mathfrak{f}_n(x,y)$ et stable par les opération de substitution et par récursion bornée.

Théorème 17.

$$\mathcal{E}^3 = \mathcal{E}$$

Démonstration. $\mathfrak{f}_3 \in \mathcal{E}$ puisqu'on peut la définir ainsi

$$\begin{aligned} g(0,y) &= y \\ g(x+1,y) &= (g(x,y)+2)^2 \\ g(x,y) &< (y+2)^{2^{2^{2x}}} \\ \mathfrak{f}_3(x,y) &= g(2^x,y) \end{aligned}$$

On peut aussi définir la fonction $(x,y) \mapsto x^y$ grâce aux éléments de la classe $\mathcal{E}^3$.

$$\begin{aligned} x+0 &= x \\ x+(y+1) &= (x+y)+1 \\ x+y &\leqslant \mathfrak{f}_3(x,y) \\ x0 &= 0 = \pi_2(x,0) \\ x(y+1) &= xy+x \\ xy &\leqslant \mathfrak{f}_3(x,y) \\ x^0 &= 1 \\ x^{y+1} &= x^y x \\ x^y &\leqslant f_3(x,y) \end{aligned}$$

Ainsi les fonctions initiales de $\mathcal{E}^3$ appartiennent à $\mathcal{E}$ et réciproquement. Et comme ces classes sont closes par les mêmes opérations, on a bien $\mathcal{E} = \mathcal{E}^3$. □

Proposition 75.

$\mathcal{E}^0$ contient les fonctions de couplage $x \mapsto Qx = Ex = x \dot{-} \lfloor \sqrt{x} \rfloor^2$, R est Q^n et est clos par minimisation bornée.

Démonstration. On peut définir dans cette classe, les fonctions :

— La fonction nulle

$$(x \mapsto O(x)) := \pi_2(x,0)$$

— La première projection

$$((x,y,z) \mapsto \pi_1(x,y,z)) := \pi_1(x,\pi_1(y,z))$$

— La seconde projection

$$((x,y,z) \mapsto \pi_2(x,y,z)) := \pi_2(x, \pi_1(y,z))$$

— La troisième projection

$$((x,y,z) \mapsto \pi_3(x,y,z)) := \pi_2(x, \pi_2(y,z))$$

— Le prédécesseur

$$(x \mapsto P(x) = x \dot{-} 1) := \begin{cases} P(0) = O(0) \\ P(x+1) = \pi_1(x, P(x)) \\ P(x) \leqslant \pi_1(x,x) \end{cases}$$

— La soustraction naturelle

$$((x,y) \mapsto x \dot{-} y) := \begin{cases} x \dot{-} 0 = \pi_1(x,x) \\ x \dot{-} (y+1) = \pi_3(x,y,P(x \dot{-} y)) \\ x \dot{-} y \leqslant \pi_1(x,y) \end{cases}$$

—

$$((x,y) \mapsto \sigma(x,y) = x \cdot 0^y = x(1 \dot{-} y)) := \begin{cases} \sigma(x,0) = \pi_1(x,x) \\ \sigma(x,y+1) = 0 \\ \sigma(x,y) \leqslant \pi_1(x,y) \end{cases}$$

—

$$((x,y) \mapsto \tau(x,y) = x + 0^y) := \begin{cases} \tau(x,0) = x+1 \\ \tau(x,y+1) = \pi_1(x,y,\tau(x,y)) \\ \tau(x,y) \leqslant \pi_1(x,y) + 1 \end{cases}$$

— Le modulo

$$((x,y) \mapsto r(x,y) = x\%y) := \begin{cases} r(0,y) = O(y) \\ r(x+1,y) = \pi_2(x, \sigma(r(x,y)+1, \\ \qquad 1 \dot{-} (y \dot{-} (r(x,y)+1)))) \\ r(x,y) \leqslant \pi_2(x,y) \end{cases}$$

—

$$\begin{cases} \left\lfloor \frac{0}{y} \right\rfloor = O(y) \\ \left\lfloor \frac{x+1}{y} \right\rfloor = \tau\left(\left\lfloor \frac{x}{y} \right\rfloor, r(x+1,y) \right) \\ \left\lfloor \frac{x}{y} \right\rfloor \leqslant \pi_1(x,y) \end{cases}$$

—

$$\begin{cases} \left\lfloor \sqrt{0} \right\rfloor = 0 \\ \left\lfloor \sqrt{x+1} \right\rfloor = \tau\left(\left\lfloor \sqrt{x} \right\rfloor, (\left\lfloor \sqrt{x} \right\rfloor + 1) \dot{-} \left\lfloor \frac{x+1}{\left\lfloor \sqrt{x} \right\rfloor + 1} \right\rfloor \right) \\ \left\lfloor \sqrt{x} \right\rfloor \leqslant \pi_1(x,x) \end{cases}$$

—

$$\begin{cases} E(0) = 0 \\ E(x+1) = \sigma\left(E(x) + 1, 1 \dot{-} r\left(x+1, \left\lfloor \sqrt{x+1} \right\rfloor\right)\right) \\ E(x) \leqslant \pi_1(x,x) \end{cases}$$

—

$$Qx = Ex$$

—

$$\begin{cases} Q^0(x) = x \\ Q^{n+1}(x) = Q(Q^n(x)) \\ Q^n(x) \leqslant \pi_1(x,n) \end{cases}$$

—

$$\begin{cases} W(0,y) = O(y) \\ W(x+1,y) = \tau(W(x,y), 2 \dot{-} \tau(1 \dot{-} (y \dot{-} Qx), Qx \dot{-} y)) \\ W(x,y) \leqslant \pi_1(x,y) \end{cases}$$

La fonction W satisfait

$$\begin{aligned} W(x+1,y) &= W(x,y) + 0^{Qx \dot{-} y} \\ Rx &= W(x,Qx) \end{aligned}$$

Les fonction Q et R sont des fonctions de couplage. La fonction Q atteint chaque entier une infinité de fois. La fonction R indique pour combien d'entier $s < x$, l'égalité $Qx = Qs$ est vérifiée.

Supposons maintenant que F appartient à $\mathcal{E}^0$. Par conséquent, $\mathcal{E}^0$ contient aussi la fonction définie par

$$\begin{aligned} f(u,0) &= 1 \dot{-} F(u,0) \\ f(u,x+1) &= \tau(f(u,x), F(u,x+1)) \\ f(u,x) &\leqslant \pi_2(u,x+2) \end{aligned}$$

On peut aussi la voir comme

$$f(u,x) = \sum_{i \in [\![0,x]\!]} (1 \dot{-} F(u,i))$$

Cette restriction de somme ne sort donc pas de $\mathcal{E}^0$. Donc $\mathcal{E}^0$ est stable par minimisation bornée. □

Proposition 76.

Pour tout $n \in \mathbb{N}$, $\mathcal{E}^n$ est stable par la minimisation bornée. L'ensemble des relations de la clesse $\mathcal{E}^n$ est stable par les opération du calcul propositionnel.

Démonstration. Les opérations définies précédemment valent également dans $\mathcal{E}^n$ puisque $\mathcal{E}^0 \subseteq \mathcal{E}^n$. □

Proposition 77.

$$\mathcal{E}^n \subseteq \mathcal{E}^{n+1}$$

Démonstration. Pour cette preuve, on commence par montrer un lemme.

Lemme 11.

$$\forall n \in \mathbb{N}^*, \forall (x,y) \in \mathbb{N}^2, \mathfrak{f}_{n+1}(x,y) > \mathfrak{f}_n(x,y)$$

Démonstration. Par récurrence : pour $n = 1$

$$\mathfrak{f}_3(x,y) = (x+1)(y+1) > x + y = \mathfrak{f}_1(x,y)$$

Pour $n \geqslant 2$. Si $x = 0$ on a l'inégalité

$$\mathfrak{f}_{n+1}(0,y) = \mathfrak{f}_n(y+1, y+1) > \mathfrak{f}_n(0,y)$$

Par suite pour tout y on a

$$\mathfrak{f}_{n+1}(k,y) > \mathfrak{f}_n(k,y)$$

D'où

$$\begin{aligned} \mathfrak{f}_{n+1}(k+1,y) &= \mathfrak{f}_{n+1}(k, \mathfrak{f}_{n+1}(k,y)) \\ &> \mathfrak{f}_{n+1}(k, \mathfrak{f}_n(k,y)) \\ &> \mathfrak{f}_n(k, \mathfrak{f}_n(k,y)) \\ &> \mathfrak{f}_n(k+1,y) \end{aligned}$$

on obtient le lemme lemme par récurrence. □

Supposons maintenant que pour $n > 2$ et $i < n$ la fonction $\mathfrak{f}_i$ est dans la classe $\mathcal{E}^{n+1}$ nous allons montrer que $\mathfrak{f}_{i+1} \in \mathcal{E}^{n+1}$. La fonction $\mathfrak{f}_{i+1}$ satisfait les conditions

$$\begin{aligned}\mathfrak{f}_{i+1}(0,y) &= \mathfrak{f}_i(y+1,y+1)\\ \mathfrak{f}_{i+1}(x+1,y) &= \mathfrak{f}_{i+1}(x,\mathfrak{f}_{i+1}(x,y))\\ \mathfrak{f}_{i+1}(x,y) &\leqslant \mathfrak{f}_{n+1}(x,y)\end{aligned}$$

La fonction $\mathfrak{f}_{i+1}$ peut être définie dans $\mathcal{E}^{n+1}$ en terme de minimisation, d'une façon similaire à celle dans laquelle les fonctions satisfaisant les conditions de la récursion bornée dont définies grâce à la minimisation, à savoir en utilisant les suites de nombres premiers. La différence est qu'ici ont utilise une paire de suites de nombres premiers, définie comme $p(x,y) = p_{P(x,y)}$ où $P(x,y)$ est une fonction de couplage.

On a

$$\begin{aligned}F(x,y) &= p(x,\mathfrak{f}_{n+1}(x,y)^{(x+2)\mathfrak{f}_{n+1}(x+y)})\\ \mathfrak{f}_{i+1}(x,y) &= MinB\left(z \mapsto \sum_{m\in[\![0,F(x,y)]\!]}\left[z+1 = \exp\left(m,p\left(x,y\right)\right)\right.\right.\\ &\qquad \prod_{v\in[\![0,m]\!]}\left[\exp\left(m,p\left(0,v\right)\right)\neq 0 \Rightarrow \exp\left(m,p\left(0,v\right)\right)\right.\\ &\qquad\quad = \mathfrak{f}_i\left(v+1,v+1\right)+1]\\ &\qquad \prod_{(i,v,w)\in[\![0,m]\!]^3}[t\neq 0][t = \exp(m,p(w+1,v))\\ &\qquad\quad \left.\left.\Rightarrow t = \exp\left(m,p\left(w,\exp\left(m,p\left(w,v\right)\right)\dot{-}1\right)\right)\right]\right]\left(\mathfrak{f}_{n+1}\left(x,y\right)\right)\end{aligned}$$

Le m utilisé dans la définition précédente a la propriété suivante : le nombre premier $p(w,v)$ apparait dans la décomposition de m en facteurs premiers avec un exposant strictement positif si et seulement si $\mathfrak{f}_{i+1}(w,v)$ apparait dans le calcul de $\mathfrak{f}_{i+1}(x,y)$ selon la définition récursive donnée précédemment. On peut aussi montrer que si $\exp(m,p(w,u)) \neq 0$ alors

$$\exp(m,p(w,u)) = \mathfrak{f}_{i+1}(w,u)+1$$

Par conséquent, si $\mathfrak{f}_i \in \mathcal{E}^{n+1}$ alors $\mathfrak{f}_{i+1} \in \mathcal{E}^{n+1}$ pour tout $i \leqslant n$. Pour tout $i \leqslant n$ on peut définir $\mathfrak{f}_i$ dans la classe $\mathcal{E}^{n+1}$. Par conséquent, pour tout $i \leqslant n$, $\mathcal{E}^i \subseteq \mathcal{E}^{n+1}$. □

Définition 66.

Pour tout $n \in \mathbb{N}$, on note $\mathcal{W}_1^n$ le plus petit ensemble de fonction contenant $x \mapsto x^2$, K, L, $x \mapsto \mathfrak{f}_n(Kx, Lx)$ et stable par les opération de composition et somme.

Cette classe de fonction est définie par induction. Pour chaque fonction f de cette classe, on peut donner un arbre dont la racine est la fonction f, les feuilles sont des fonctions de base et chaque nœud s'obtient en appliquant un schéma de construction à ses fils. L'ordre de f est la profondeur minimale des arbres qui décrivent la construction de f. L'ordre d'une fonction dépend donc de la classe dans laquelle on considère la fonction.

Proposition 78.

Soit $n \in \mathbb{N} \setminus \{0, 1\}$. Soit f une fonction d'ordre k dans $\mathcal{W}_1^n$. On a

$$f(x) < \mathfrak{f}_{n+1}(k, x)$$

Démonstration. On fait une preuve par induction sur les arbres précédemment décrits.

On commence par les fonctions de base.

$$\begin{aligned} x^2 < (x+2)^2 = \mathfrak{f}_3(0,x) &\leqslant \mathfrak{f}_{n+1}(0,x) \\ Kx \leqslant x &< \mathfrak{f}_{n+1}(0,x) \\ Lx \leqslant x &< \mathfrak{f}_{n+1}(0,x) \\ \mathfrak{f}_n(Kx, Lx) \leqslant \mathfrak{f}_n(x,x) &< \mathfrak{f}_n(x+1, x+1) = \mathfrak{f}_{n+1}(0,x) \end{aligned}$$

Supposons que g et h sont des fonctions d'ordre respectifs l et k dans $\mathcal{W}_1^n$ et satisfont la proposition. On suppose que $k \geqslant l$. On a

$$\begin{aligned} g(x) &< \mathfrak{f}_{n+1}(k,x) \\ h(x) &< \mathfrak{f}_{n+1}(k,x) \end{aligned}$$

Donc on obtient

$$\begin{aligned} g(h(x)) &< \mathfrak{f}_{n+1}(k, h(x)) \\ &< \mathfrak{f}_{n+1}(k, \mathfrak{f}_{n+1}(k,x)) \\ &= \mathfrak{f}_{n+1}(k+1, x) \end{aligned}$$

et

$$\begin{aligned} g(x) + h(x) &< 2\mathfrak{f}_{n+1}(k, x) \\ &< (\mathfrak{f}_{n+1}(k, x) + 2)^2 \\ &= \mathfrak{f}_3(0, \mathfrak{f}_{n+1}(k, x)) \\ &\leqslant \mathfrak{f}_{n+1}(k+1, x) \end{aligned}$$

Par conséquent, si $\mathfrak{f}$ est une fonction d'ordre $k+1$ et est obtenu à partir à partir des fonctions g et h grâce à un seul constructeur alors

$$f(x) < \mathfrak{f}_{n+1}(k+1, x)$$

D'où la preuve par induction. □

Proposition 79.

La fonction $x \mapsto \mathfrak{f}_{n+1}(x, x)$ croit plus vite que toute fonction de la classe $\mathcal{E}^n$.

Démonstration. Pour $n \geqslant 2$, il vient du théorème précédent que si f est l'ordre k dans $\mathcal{W}_1^n$ et $x \geqslant k$ alors

$$f(x) < \mathfrak{f}_{n+1}(x, x)$$

D'où $x \mapsto \mathfrak{f}_{n+1}(x, x)$ croit plus vite que toute fonction de la classe $\mathcal{W}_1^n$ et croit donc plus vite que toute fonction de la classe $\mathcal{E}^n$ puisque les fonctions initiales de la classe $\mathcal{E}^n$ sont dominées par la fonction strictement croissante $(x, y) \mapsto \mathfrak{f}_n(x+1, y+1)$ de la classe $\mathcal{W}_1^n$.

Pour $n \in \{0, 1\}$, on peut vérifier directement le théorème. On peut prouver que si f est d'ordre k dans $\mathcal{E}^0$, alors $f(x) < x + 2^k + 1$. De même, si f est d'ordre k dans $\mathcal{E}^1$ alors $f(x) < (x+1)2^k$ Donc la fonction $x \mapsto 2x$ croit plus vite que toute fonction $f \in \mathcal{E}^0$ et $x \mapsto (x+1)^2$ croit plus vite que toute fonction $f \in \mathcal{E}^1$. □

Corollaire 18.

$$\mathcal{E}^n \subsetneq \mathcal{E}^{n+1}$$

Théorème 18.

$$\bigcup_{n \in \mathbb{N}} \mathcal{E}^n = \mathcal{PR}$$

Démonstration. Comme les fonctions de bases et les schémas de construction des classes $\mathcal{E}^n$ sont primitifs récursifs, les classes $\mathcal{E}^n$ sont des sous ensemble des fonctions primitives récursives. D'où

$$\bigcup_{n\in\mathbb{N}} \mathcal{E}^n \subseteq \mathcal{PR}$$

Lemme 12.

Si $n \geqslant 2$ et $f \in \mathcal{E}^n$ alors il y a un entier m tel que

$$f(x_1, \ldots, x_k) \leqslant \mathfrak{f}_{n+1}{}^m(\max(x_1, \ldots, x_k))$$

Démonstration. La preuve est aisée par induction sur l'arbre décrivant f. □

Lemme 13.

Si $f \in \mathcal{E}^n$ alors l'itérée $y \mapsto f^y \in \mathcal{E}^{n+1}$.

Démonstration. On suppose par exemple que f a deux variables et que l'itération porte sur la première. On a

$$f(x, y) \leqslant E^m_{n-1}(\max(x, y))$$

et par récurrence

$$f^z(x, y) \leqslant E^{mz}_{n-1}(\max(x, y))$$

pour

$$\begin{aligned} f^0(x, y) &= x \\ &\leqslant \max(x, y) \\ &= E^0_{n-1}(\max(x, y)) \end{aligned}$$

et

$$\begin{aligned} f^{z+1}(x, y) &= f(f^z(x, y)) \\ &\leqslant E^m_{n-1}(\max(E^{mz}_{n-1}(\max(x, y)), y)) \\ &\leqslant E^{m(z+1)}_{n-1}(\max(x, y)) \end{aligned}$$

Mais aussi

$$E^{mz}_{n-1}(\max(x, y)) \leqslant E_n(mz + \max(x, y))$$

aussi, en combinant ces deux dernières inégalités on voit que $y \mapsto f^y$ peut être défini par récursion bornée dans $\mathcal{E}^{n+1}$.

On traite indépendamment et aisément les cas où $n \in \{0, 1\}$. □

Corollaire 19.

Si g et h appartiennent à $\mathcal{E}^n$ et $f = Rec(g,h)$, alors $f \in \mathcal{E}^{n+1}$.

Démonstration. On utilise le lemme précédent. □

Ainsi la hiérarchie de GRZEGORCZYK est stable par récursion primitive. Aussi, on a $\mathcal{PR} \subseteq \bigcup_{n\in\mathbb{N}} \mathcal{E}^n$.

Par conséquent $\bigcup_{n\in\mathbb{N}} \mathcal{E}^n = \mathcal{PR}$. □

Chapitre 19

La hiérarchie arithmétique

19.1 Définition

Définition 67.

Une formule est dite prénexe si elle est constitué d'une suite de quantificateurs suivie d'une formule ne contenant que des quantificateurs bornés. On a donc une formule de la forme

$$Q_i x_i \ \dots \ Q_n x_n \, \rho$$

où les Q_i sont des quantificateurs et ρ est une formule à quantificateurs bornés.

On va définir la hiérarchie arithmétique qui est une classification des formules prénexes.

Définition 68 (Hiérarchie arithmétique des formules prénexes)**.**

$\Pi_0 = \Sigma_0$ est l'ensemble des formules à quantificateur borné.
Soit S une formule de Σ_n et P une formule de Π_n. On note x une variable quelconque (libre ou non dans S et P). On a :

- $\exists x : S$ est une formule Σ_n
- $\exists x : P$ est une formule Σ_{n+1}
- $\forall x, S$ est une formule Π_{n+1}
- $\forall x, P$ est une formule Π_n

Définition 69 (Hiérarchie arithmétique des parties de $\mathbb{N}$)**.**

Un sous-ensemble de $\mathbb{N}$ est dit Σ_n (resp. Π_n) s'il peut être défini par une formule Σ_n (resp. Π_n) à une variable libre.

Définition 70.

Un ensemble à la fois Σ_n et Π_n est dit Δ_n.

Il peut être pratique d'élargir la définition aux n-uplets d'entiers.

Définition 71 (Hiérarchie arithmétique des parties de $\mathbb{N}^p$)**.**

Un sous ensemble de $\mathbb{N}^p$ est dit Σ_n (resp. Π_n) s'il peut être défini par une formule Σ_n (resp. Π_n) à p variable libre.

19.2 Propriétés

Proposition 80.

La négation d'une formule Σ_n est Π_n, et réciproquement.

Corollaire 20.

Le complémentaire d'un ensemble Σ_n est Π_n, et réciproquement

Corollaire 21.

Δ_n est stable par complémentation.

Proposition 81.

$$\forall n, \Sigma_n \cup \Pi_n \subsetneq \Sigma_{n+1} \cap \Pi_{n+1} = \Delta_{n+1}$$

Proposition 82.

Pour tout n, les classes Σ_n, Π_n et Δ_n sont stables par intersection, réunion et produit cartésien.

19.3 Théorème de POST

Le théorème de POST donne une relation entre les degrés de TURING et la hiérarchie arithmétique.

Théorème 19 (POST)**.**

Soit B un ensemble.
B est Σ_{n+1} si et seulement si B est récursivement énumérable par une machine de TURING à oracle sur $\emptyset^{(n)}$.

Quatrième partie

Des systèmes TURING-complets

Chapitre 20

Des langages de programmation TURING-complets

20.1 Généralités à propos de l'assembleur

Il s'agit de justifier le fait que les ordinateurs ne peuvent pas être plus puissants que les machines de TURING. Il est nécessaire pour cela de détailler le fonctionnement de l'exécution d'un programme.

Lors du développement d'un programme, on écrit un code dans un langage de programmation. Lors de la compilation, opération qui fait un programme à partir d'un code, il y a trois étapes. Dans un premier temps, le compilateur résout les inclusions puis chaque instruction est transformée en une suite d'instructions en assembleur. Enfin ce programme est simplement transcrit en langage machine selon une table d'équivalence.

L'assembleur est un langage minimaliste ne décrivant que des opérations élémentaires comme celles des machines RAM. On peut se convaincre que les instructions des machines RAM suffisent à simuler les instructions existant en assembleur. En effet, une très large majorité des instructions sont des opérations arithmétiques (ou peuvent être décrites ainsi) ou des sauts conditionnels (utilisant éventuellement des opérations arithmétiques).

Ainsi un ordinateur ou un langage de programmation ne peut pas avoir une puissance de calcul supérieure à celle des machines de TURING. C'est pourquoi, pour un langage de programmation, on ne parle pas de TURING-équivalence mais de TURING-complétude. Bien qu'en informatique ces deux notions soient en pratique équivalentes, formellement la TURING-complétude n'assure qu'une puissance au moins égale à celle des machines de TURING.

20.2 Brainfuck

La description du Brainfuck a déjà été donnée lors de la démonstration de la TURING-équivalence du λ-calcul. Il s'agit de montrer que ce langage est TURING-équivalent pour valider formellement la démonstration de la TURING-équivalence du λ-calcul.

Le Brainfuck est évidemment simulable par une machine RAM. Le Brainfuck n'est donc pas plus puissant qu'une machine de TURING. Ce résultat était prévisible d'après les commentaires faits sur l'assembleur. Réciproquement, il est facile de simuler, grâce à ce langage, des machines à compteurs. Ainsi ce langage est effectivement TURING-complet.

20.3 Haskell

Le Haskell a été utilisé dans la démonstration de la TURING-équivalence du λ-calcul. On sait désormais que ce langage est TURING-complet puisque grâce à lui on a simulé du Brainfuck qui est lui-même TURING-complet. Il s'agit de montrer qu'il est aussi équivalent au λ-calcul.

En réalité on n'a pas besoin de l'équivalence du langage entier mais seulement des fonctions et mots-clefs utilisés. On aura ainsi l'équivalence du λ-calcul avec les machines de TURING, ce qui impliquera que le langage entier est équivalent au λ-calcul.

On utilise la fonction `length` qui peut de définir ainsi :

```
1 length []=0
2 length (t:q)=1+length q
```

Ce qui est équivalent à :

$$\begin{aligned}\text{length} &= \lambda l.\,\text{liste\_it}\ l\ (\lambda y\,m.\,\text{add}\ (\lambda f\,x.f\,x)\,m)\ (\lambda f\,x.x)\\ &= \lambda l.l\ (\lambda y\,m.\,\text{add}\ 1\,m)\,0\end{aligned}$$

où

$$\text{liste\_it} = \lambda l\,x\,m.l\,x\,m$$

On vérifie que cette fonction fait bien ce qu'on attends d'un itérateur de liste

$$\begin{aligned}&\text{liste\_it}\,[]\ (\lambda a\,b.f\,a\,b)\,v = v\\ &\text{liste\_it}\ l\ (\lambda a\,b.f\,a\,b)\,v = f\ (\text{head}\ l)\ (\text{liste\_it}\,(\text{tail}\ l)\ (\lambda a\,b.f\,a\,b)\,v)\end{aligned}$$

ou, de façon plus explicite

$$\text{liste\_it}\,[a_1,a_2,\ldots,a_n]\ (\lambda a\,b.f\,a\,b)\,v = f\ a_1\ (f\ a_2(\ldots(f\ a_n\ v)\ldots))$$

On vérifie également que la définition de length est cohérente. D'abord dans le cas d'une liste vide

$$\begin{aligned}\text{length}\,[] &= \text{liste\_it}\,[]\ (\lambda y\,m.\,\text{add}\ (\lambda f\,x.f\,x)\,m)\ (\lambda f\,x.x)\\ &= \text{liste\_it}\ \lambda g\,y.y\ (\lambda y\,m.\,\text{add}\ (\lambda f\,x.f\,x)\,m)\ (\lambda f\,x.x)\\ &= (\lambda l\,x\,m.l\,x\,m)\ (\lambda g\,y.y)\ (\lambda y\,m.\,\text{add}\ (\lambda f\,x.f\,x)\,m)\ (\lambda f\,x.x)\\ &= (\lambda g\,y.y)\ (\lambda y\,m.\,\text{add}\ (\lambda f\,x.f\,x)\,m)\ (\lambda f\,x.x)\\ &= \lambda f\,x.x\\ &= 0\end{aligned}$$

puis dans le cas d'une liste non vide

$$\begin{aligned}
\text{length}(\text{Cons } A\, B) &= (\lambda l.\,\text{liste_it } l\ (\lambda y\, m.\,\text{add } 1\, m)\, 0)\ ((\lambda a\, b\, p.p\, a\, b)\, A\, B) \\
&= \textit{liste_it}\ (\lambda f\ x.f\ A\ (B\, f\, x))\ (\lambda y\, m.\,\text{add } 1\, m)\, 0 \\
&= (\lambda l\, x\, m.l\, x\, m)\ (\lambda f\ x.f\ A\ (B\, f\, x))\ (\lambda y\, m.\,\text{add } 1\, m)\, 0 \\
&= (\lambda f\ x.f\ A\ (B\, f\, x))\ (\lambda y\, m.\,\text{add } 1m)\, 0 \\
&= (\lambda y\, m.\,\text{add } 1\, m)\ A\ (B(\lambda y\, m.\,\text{add } 1\, m)\ (\lambda f\ x.x)) \\
&= (\lambda y\, m.\,\text{add } 1\, m)\ A\ (B\ (\lambda y\, m.\,\text{add } 1\, m)\, 0) \\
&= \text{add } 1\ (B\ (\lambda y\, m.\,\text{add } 1\, m)\, 0) \\
&= \text{add } 1\ (\text{length } B)
\end{aligned}$$

```haskell
assoc ((a,b):q) e
    |a==e = b
    |otherwise = assoc q e
```

Cette fonction pose comme seul problème le test d'égalité. On le résout grâce à sub. On rappelle que sub $n\, p$ donne $n - p$ si $n \geqslant p$ et 0 sinon. Le test d'égalité devient

$$\text{eq} = \text{et}\ (\text{isZero}\ (\text{sub}\ n\, p))\ (\text{isZero}\ (\text{sub}\ p\, n))$$

où

$$\text{et} = \lambda a\, b.\,\text{ifThenElse}\ a\, b\ \text{false}$$

D'où

$$\text{assoc} = \lambda l\, e.\,\text{liste\_it}\ l\ (\lambda a\, b.\,\text{ifThenElse}\ (\text{eq}\ (\text{first}\ a)\, e)\ (\text{second}\ a)\, b)\, 0$$

```haskell
reciproque ((a,b):q) e --Idem
    |b==e = a
    |otherwise = reciproque q e
```

Cette fonction ne pose pas de problème :

$$\text{reciproque} = \lambda l, e.\,\text{liste\_it}\ l\ (\lambda a\, b.\,\text{ifThenElse}\ (\text{eq}\ (\text{second}\ a)\, e)\ (\text{first}\ a)\, b)\, 0$$

On doit aussi montrer qu'il est possible d'accéder à n'importe quel élément d'une liste :

```haskell
at l n --Accede un element arbitraire d'index donne dans une liste
    |n==0 = head l
    |otherwise = at (tail l) (n-1)
```

$$\text{at} = \lambda l\, n.\, \text{liste\_it}\ l\ (\lambda a\, b.\, \text{ifThenElse}\ (\text{isZero}\ n)\ a\ b)\ 0$$

Les autres fonctions n'utilisent rien qui ait été précédemment défini, donc peuvent être écrites en λ-calcul.

Ainsi les fonctions décrites ici en Haskell existent bien en λ-calcul.

Plus généralement, le Haskell est un langage purement fonctionnel qui a pour but de rester très proche des considérations fonctionnelles du λ-calcul.

20.4 Whitespace

Le WhiteSpace est un langage impératif, exotique et qui utilise une mémoire organisée en une pile.

20.4.1 Description du langage

Les seuls symboles utilisés pour la programmation sont l'espace ([Space]), la tabulation ([Tab]) et le retour à la ligne ([LF]). Chaque instruction consiste en une suite d'unités lexicales dont le premier est nommé IMP (Instruction Modification Parameter).

IMP	Catégorie
[Space]	Opération sur la pile
[Tab] [Space]	Opérations arithmétiques
[Tab] [Tab]	Accès au tas
[LF]	Contrôles de flux
[Tab] [LF]	Opération d'entrée/sortie

La machine virtuelle qui exécute les programmes a une pile et un tas. Le programmeur est libre d'empiler des entiers arbitrairement grands sur la pile. Le tas peut aussi être lu comme une mémoire permanente de variables et de structures de données.

Certaines commandes prennent un paramètre sous forme d'un label ou d'un entier. Les entiers peuvent être arbitrairement grands et sont simplement représentés bit-à-bit sous la forme d'une suite de [Space]et [Tab]suivie d'un [LF]. [Space]et [Tab]représentent respectivement les chiffres 0 et 1. Le signe de l'entier est donné par le premier caractère : [Space]pour un signe positif et [Tab]pour un signe négatif. Un label est simplement une suite de [Space]et [Tab]terminée par un [LF]. Toutes les variables sont globales et il n'y a qu'un namespace donc chaque label doit être unique.

Les opérations sur la pile

Les opérations sur la pile sont les opérations les plus fréquentes, ce qui explique la petitesse de leur IMP : [Space]. Il y a 4 instructions sur la pile :

Instruction	Paramètres	Effet
[Space]	Entier	Empile l'entier
[LF] [Space]	-	Duplique l'objet au sommet de la pile
[LF] [Tab]	-	Échange les deux objets du dessus de la pile
[LF] [LF]	-	Dépile le sommet

Les opérations arithmétiques

Les opérations arithmétiques agissent sur les deux éléments au sommet de la pile et les remplacent par le résultat du calcul. Le premier élément empilé est considéré comme étant à gauche de l'opérateur.

Instruction	Paramètres	Effet
[Space] [Space]	-	Addition
[Space] [Tab]	-	Soustraction
[Space] [LF]	-	Multiplication
[Tab] [Space]	-	Division d'entiers
[Tab] [Tab]	-	Modulo

Les accès au tas

Les instructions d'accès au tas lisent la pile pour trouver l'adresse des objets à stocker ou à lire. Pour stocker un objet, on empile l'adresse puis la valeur et on exécute la commande de stockage. Pour lire un élément, on empile l'adresse et on lance la commande de lecture, laquelle empile la valeur trouvée.

Instruction	Paramètres	Effet
[Space]	-	Enregistrement
[Tab]	-	Lecture

Les contrôles de flux

Les contrôles de flux sont aussi des instructions fréquemment utilisées. Les sous-routines sont marquées par des labels, qui sont la destination de sauts conditionnels ou inconditionnels, grâce auxquels les boucles peuvent être implémentées. Les programmes doivent finir par [LF] [LF] [LF] qui est l'instruction pour finir le programme.

On note s le sommet de la pile

Instruction	Paramètres	Effet
[Space] [Space]	Label	Déclare un label dans le programme
[Space] [Tab]	Label	Appelle une sous-routine
[Space] [LF]	Label	Saute à un label
[Tab] [Space]	Label	Saute à un label si $s = 0$
[Tab] [Tab]	Label	Saute à un label si $s < 0$
[Tab] [LF]	-	Termine une sous-routine
[LF] [LF]	-	Termine le programme

Les opérations d'entrée/sortie

Enfin, on a besoin de pouvoir interagir avec l'utilisateur. Il y a des opérations d'écriture et de lecture pour les nombres et les caractères. Avec ça on peut créer des routines pour manipuler les chaînes de caractères.

Les fonctions de lecture prennent l'adresse de stockage du résultat au sommet de la pile.

On note s le sommet de la pile

Instruction	Paramètres	Effet
[Space] [Space]	-	Renvoie s comme un caractère
[Space] [Tab]	-	Renvoie s comme un nombre
[Tab] [Space]	-	Enregistre un caractère lu à l'adresse s
[Tab] [Tab]	-	Enregistre un nombre lu à l'adresse s

20.4.2 TURING-complétude

La TURING-complétude de ce langage n'est pas évidente puisqu'on a une structure de mémoire organisée en une pile. Il serait alors légitime de se dire que ce langage ne peut pas être beaucoup plus puissant que les automates à une pile définis plus loin.

Cependant il s'avère que ce langage est TURING-complet. On peut en effet simuler les machines à deux compteurs grâce à un programme en WhiteSpace. On choisit pour cela d'utiliser une pile de taille constante contenant que 2 entiers. Le sommet de la pile représente le compteur noté C_0 et le deuxième élément représente le compteur C_1.

L'incrémentation de C_0 ne pose pas de problème. On commence par empiler 1 puis on additionne les deux éléments du dessus de la pile :

1. [Space] [Space] [Space] [Tab] [LF] Empilage de 1
2. [Tab] [Space] [Space] [Space] Remplacement des deux éléments du dessus de la pile par leur somme

La décrémentation est similaire :

1. [Space] [Space] [Tab] [Tab] [LF] Empilage de -1
2. [Tab] [Space] [Space] [Space]

L'incrémentation de C_1 demande au préalable de permuter les deux compteurs :

1. [Space] [LF] [Tab] Permutation des deux compteurs

2. [Space] [Space] [Space] [Tab] [LF] Empilage de 1
3. [Tab] [Space] [Space] [Space] Remplacement des deux éléments du dessus de la pile par leur somme
4. [Space] [LF] [Tab] Permutation des deux compteurs

On utilise le même procédé pour la décrémentation de C_1

1. [Space] [LF] [Tab]
2. [Space] [Space] [Tab] [Tab] [LF] Empilage de -1
3. [Tab] [Space] [Space] [Space]
4. [Space] [LF] [Tab]

Le saut conditionnel sur le compteur C_0 se fait en demandant un saut conditionnel puis en réalisant un saut conditionnel qui n'est atteint que si l'instruction n'a pas été exécuté. On suppose pour cela que chaque instruction est précédé par un label qui lui est propre. On utilisera donc les numéros des lignes pour les sauts plutôt que les labels qu'on n'explicitera pas.

1. [LF] [Tab] [Space] (Label 1) Saute au Label 1 si C_0 est nul
2. [LF] [Space] [LF] (Label 2) Saute au Label 2. Cette instruction est atteinte si l'instruction précédente n'est pas exécuté.

Pour le saut conditionnel sur C_1 demande des échanges des compteurs ce qui pose quelques problèmes pratiques.

1. [Space] [LF] [Tab] Permutation des deux compteurs
2. [LF] [Tab] [Space] 4 Saut conditionnel vers la ligne ici notée 4
3. [LF] [Space] [LF] 6 Saut inconditionnel vers la ligne ici notée 4
4. [Space] [LF] [Tab]
5. [LF] [Space] [LF] (Label 1) Saut conditionnel vers le Label 1
6. [Space] [LF] [Tab]
7. [LF] [Space] [LF] (Label 2) Saut conditionnel vers le Label 2

Ainsi on peut simuler les machines à deux compteurs en WhiteSpace. Les machines à deux compteurs étant équivalentes aux machines de TURING, on en déduit que le WhiteSpace est TURING-complet.

20.4.3 Causes de la TURING-complétude

Comme ce langage utilise une structure de mémoire organisée en une pile et puisqu'il est, de façon surprenante, TURING-complet, il est raisonnable de se demander quelles sont les instructions que ne peuvent pas simuler les automates à pile et qui permettent à ce langage d'atteindre la puissance des machines des TURING.

On a vu que les opérations de saut conditionnel et inconditionnel sont indispensables dans cette démonstration. Elles servent à se déplacer dans le programme afin de d'exécuter plusieurs fois un même code dans des contextes différents. Ce qui fait office de programme pour un automate à pile est le mot lu. En effet, il donne la transition a effectuer lorsque la machine est dans un état précis. Comme le mot est lu linéairement, il est évidement impossible de faire des sauts dans le mot. Pour s'affranchir de cette restriction, on peut préalablement mémoriser l'ensemble du mot et ainsi le lire dans la mémoire. On a alors besoin d'instructions permettant un accès arbitraire dans la pile.

Chapitre 21

TURING-complétude du jeu de la vie de CONWAY

On souhaite montrer l'équivalence entre les machines de TURING et les automates cellulaires c'est-à-dire l'égalité des classes des fonctions calculables par ces deux modèles de calcul.

On procède par double inclusion. On choisit un automate cellulaire particulier, le jeu de la vie de CONWAY, et on y simule une machine de TURING. On prouve ainsi que les automates cellulaires ont une puissance de calcul au moins égale aux machines de TURING. L'autre inclusion se fait en montrant qu'on peut simuler n'importe quel automate cellulaire grâce à un programme écrit dans un langage qui est au plus aussi puissant qu'une machine de TURING.

On choisit d'implémenter dans le jeu de la vie une machine de TURING universelle. Pour cela, on commence par créer une machine simple puis on montre la généralisation à une machine universelle en exhibant le résultat.

21.1 Définition du jeu de la vie de CONWAY

Le jeu de la vie pourrait être défini comme n'importe quel automate cellulaire mais cette définition serait peu appropriée. Nous allons donc tenter d'en donner une plus adaptée afin de faciliter son étude.

Le jeu de la vie des un automate cellulaire, bidimensionnel et sommatif. En tant qu'automate sommatif, on ne définit pas explicitement une règle locale mais on la remplace par deux fonctions. La première est la population dans le voisinage. La seconde associe un comportement à chaque population possible. Ce procédé peut être utilisé pour tout automate sommatif et est souvent utilisé de façon implicite. On justifie que cette façon de faire est valable en remarquant que la composée de ces deux fonctions donne la règle locale.

Définition 72 (Voisinage du jeu de la vie).

Le jeu de la vie est un automate cellulaire de dimension 2 ayant un alphabet à 2 éléments (ici $\{0,1\}$) et ayant un voisinage de MOORE appelé v. On définit également :

$$\begin{aligned} v_{Gol} : \mathbb{Z}^2 &\hookrightarrow \mathcal{P}\left(\mathbb{Z}^2\right) \\ (i,j) &\mapsto \left\{ (i+\alpha, j+\beta) \;\middle|\; (\alpha,\beta) \in [\![-1,1]\!]^2 \wedge (\alpha,\beta) \neq (0,0) \right\} \end{aligned}$$

On définit le voisinage du jeu de la vie comme un voisinage de MOORE. En fait, le jeu de la vie n'est pas rigoureusement sommatif : la transition dépend du nombre de cellules vivantes dans v_{Gol} et de la cellule elle-même. Ce qui justifie et motive la définition de v_{Gol} qui est le voisinage de MOORE privé de la cellule elle même.

On peut généraliser la notion d'automate sommatif en donnant le nombre de parties du voisinage à l'intérieur desquelles la position des cellules n'a pas d'importance. Aussi un automate 1-sommatif est simplement un automate sommatif alors qu'un automate a-sommatif, où a est l'arité de l'automate n'est absolument pas sommatif. Cela donne une indication sur le nombre de disjonctions de cas à faire pour décrire la règle locale.

Définition 73 (Configuration du jeu de la vie).

On appelle configuration du jeu de la vie toute fonction σ_n :

$$\sigma_n : \mathbb{Z}^2 \to \{0,1\}$$

Définition 74 (Population dans le voisinage du jeu de la vie).

On appelle population dans le voisinage ϵ la fonction :

$$\begin{aligned} \epsilon : \mathbb{Z} \times \mathbb{Z} \times \mathbb{N} &\to [\![0,8]\!] \\ (i,j,n) &\mapsto \left| \left\{ (a,b) \mid (a,b) \in v_{Gol}(i,j) \wedge \sigma_n(a,b) = 1 \right\} \right| \end{aligned}$$

Il s'agit du nombre de 1 dans le voisinage.

Définition 75 (Règle du jeu de la vie).

On appelle δ_{Gol} la règle locale du jeu de la Vie :

$$\begin{aligned}\delta_{Gol} : \{0,1\}^9 &\to \{0,1\} \\ (a,b) &\mapsto [|\{x \in a \mid x = 1\}| = 3] + [|\{x \in a \mid x = 1\}| = 2][b = 1]\end{aligned}$$

avec $a \in \{0,1\}^8$ et $b \in \{0,1\}$.
On définit la fonction globale d'évolution du jeu de la vie $F_{\delta_{Gol}}$ par :

$$\begin{aligned}F_{\delta_{Gol}} : \{0,1\}^{(\mathbb{Z}^2)} &\to \{0,1\}^{(\mathbb{Z}^2)} \\ \sigma_n &\mapsto \sigma_{n+1}\end{aligned}$$

où

$$\forall (i,j) \in \mathbb{Z}^2, \sigma_{n+1}(i,j) = [\epsilon(i,j,n) = 3] + [\epsilon(i,j,n) = 2] \cdot [\sigma_n(i,j) = 1]$$

Définition 76 (Jeu de la vie).

On appelle jeu de la vie l'automate cellulaire suivant :

$$(2, \{0,1\}, v, \delta_{Gol})$$

Pour se représenter le jeu de la vie de façon pratique, il faut imaginer un quadrillage indexé par $\mathbb{Z}^2$. Chaque case du quadrillage, repérée par un couple de coordonnées entières, est appelée "cellule" et peut prendre deux états : mort ou vivant. Le voisinage de chaque cellule est l'ensemble des cases adjacentes, y compris les cases adjacentes par un sommet.

Une cellule vivante ne restera vivante à la génération suivante que si elle a 2 ou 3 cellules vivantes dans son voisinage. Une cellule morte sera vivante à la génération suivante si elle a exactement 3 cellules vivantes dans son voisinage. Une génération détermine donc entièrement la suivante.

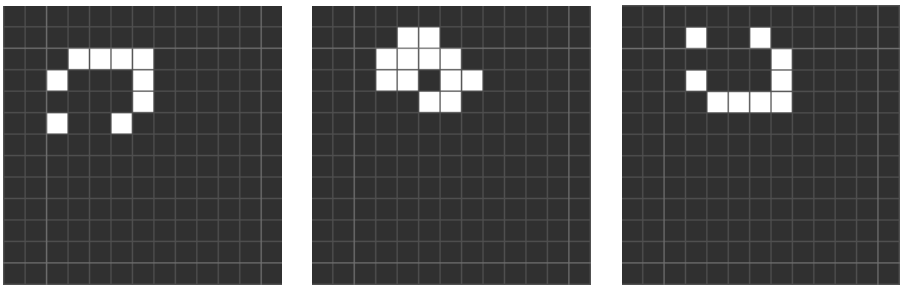
Par la suite le simulateur utilisé (Golly [RTH+12]) fait qu'une cellule vivante est représentée par une case blanche et une cellule morte par une case noire.

21.2 Structures remarquables

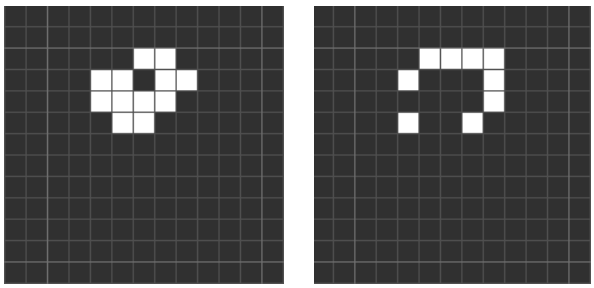
Il existe dans le jeu de la vie des structures remarquables. Nous en présentons ici quelques-unes.

LWSS

Le Light Weight Space Ship (LWSS) est un vaisseau de période 4. Son déplacement se fait dans le sens du quadrillage, à raison d'une case toutes les 2 générations.



(a) Génération 0 (b) Génération 1 (c) Génération 2



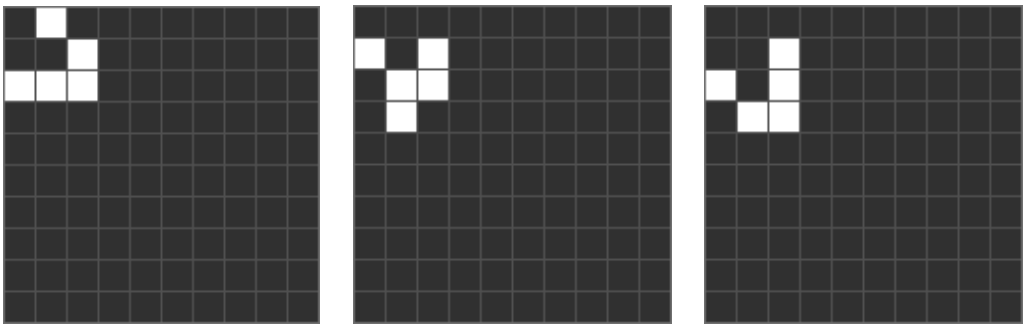
(d) Génération 3 (e) Génération 4

FIGURE 21.1 – Déplacement d'un LWSS

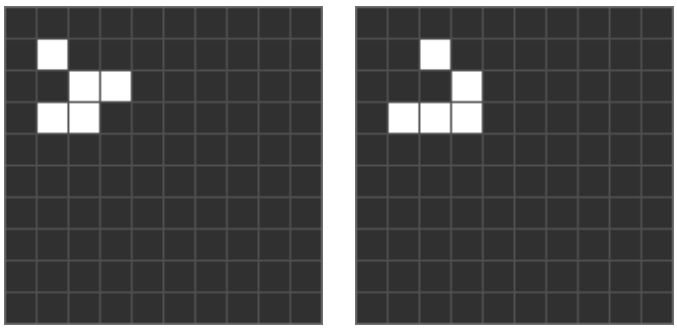
Planeur

Il s'agit d'un vaisseau de période 4. Il se trouve en effet reproduit après quatre générations à l'identique mais translaté. Il se déplace d'une case en diagonale toutes les quatre générations.

Lorsque deux planeurs se rencontrent, il y a 73 cas de collision possibles, selon leur orientation et leur déphasage.



(a) Génération 0 (b) Génération 1 (c) Génération 2

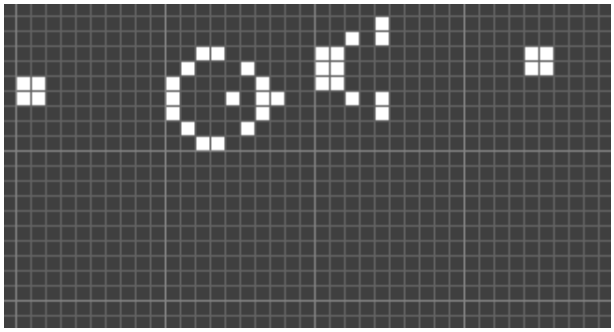


(d) Génération 3 (e) Génération 4

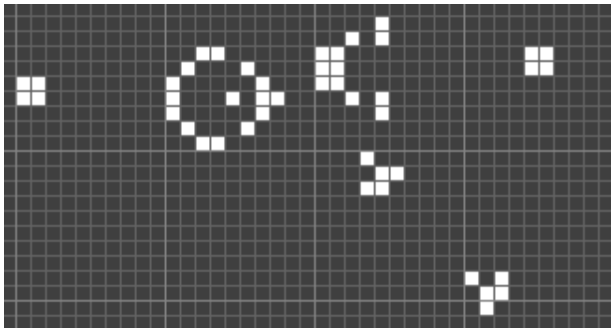
FIGURE 21.2 – Déplacement d'un planeur

Canon

On désigne par canon une structure périodique qui donne naissance à des planeurs ou tout autre vaisseau. Ce canon émet un flux de planeurs, à raison d'un planeur toutes les 30 générations. Il s'agit du canon le plus simple.



(a) Génération 0



(b) Génération 60

FIGURE 21.3 – Fonctionnement du canon

Réflecteurs

Premier modèle Le réflecteur possède lui-même une période de 14 générations, le planeur doit être bien ajusté, ici il arrive du coin en haut à gauche, il repartira dans le sens contraire.

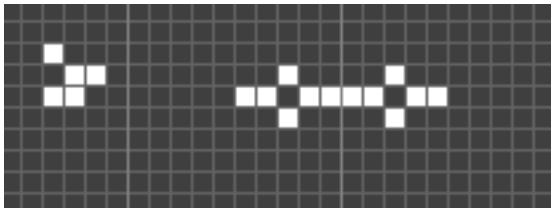


FIGURE 21.4 – Réflecteur

Second modèle Il existe un autre réflecteur dont le rôle est de changer la direction d'un flux de planeurs de 90 degrés. Il sera notamment utilisé pour retarder un flux de planeurs, en lui faisant suivre des détours. On a ici le même réflecteur avec 2 planeurs après 90 générations.

(a) Génération 0 (b) Génération 90

FIGURE 21.5 – Modèle alternatif de réflecteur

Eater

Premier modèle On reprend exactement la même structure que précédemment, mais cette fois placée verticalement. Le planeur sera détruit.

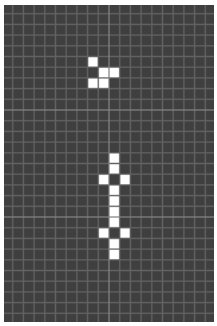
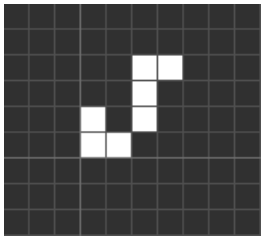


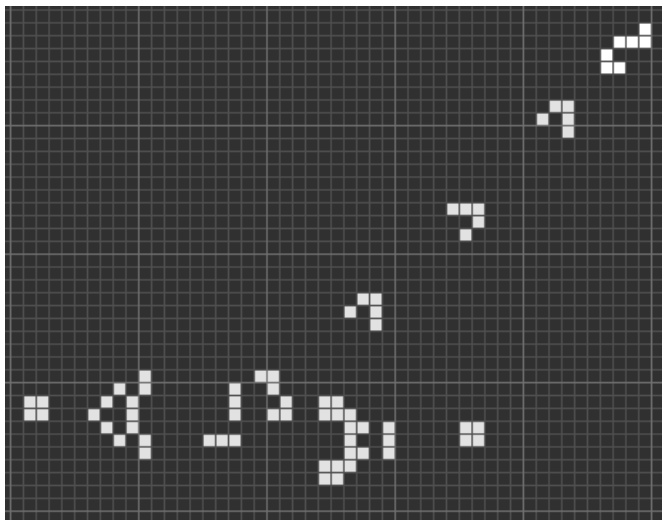
FIGURE 21.6 – Eater

Après 45 générations, on retrouve ce motif mais sans le planeur.

Un second modèle Voici un autre eater, plus répandu car plus petit, et stable. Il est ici positionné de façon à manger les planeurs issus du flux produit par le canon.



(a) Eater seul



(b) Un eater avec un canon

FIGURE 21.7 – Modèle alternatif de eater

Les planeurs créés par le canon sont détruits par le eater, situé en haut à droite.

21.3 Implémentation d'une machine simple

Nous allons maintenant implémenter une machine de TURING dans le jeu de la vie. Il s'agit de la première machine de TURING construite par Paul RENDELL. C'est également la machine la plus répandue et connue. Elle dispose d'un autre avantage : le mode de construction est généralisable et reproductible pour construire n'importe quelle machine. Nous montrerons par la suite une machine de TURING universelle construite de façon similaire. [Ren11a]

Cette machine comporte 3 symboles et 3 états (numérotés de 0 à 2). Elle duplique une série consécutive de 1 sur le ruban. Avant l'exécution, la tête de lecture se trouve sur le 1 le plus à droite. L'état initial est l'état 0.

— Elle écrit un 2, passe dans l'état 1, puis se décale vers la droite pour trouver un 0, qu'elle remplace par un 2.
— Elle passe dans l'état 0 et revient vers la gauche jusqu'à trouver un 1, écrit un 2 et prend l'état 1.
— La machine déplace sa tête de lecture vers la droite par dessus les 2, jusqu'à trouver un 0, elle écrit alors un 2 et passe dans l'état 0.
— Une fois dans l'état 0, elle déplace la tête de lecture à gauche jusqu'à trouver un 1, elle écrit alors un 2 et repasse dans l'état 0.

Elle recommence ainsi tant qu'elle peut trouver de tels nombres. Si la machine est dans l'état 0 et qu'elle trouve un 0 à gauche du ruban, c'est qu'elle a déjà trouvé tous les 1 et les a transformés en 2 et donc par suite a écrit un nombre égal de 2 à droite du ruban. Dans ce cas, elle passe dans l'état 2, dont elle ne sort pas, et en se déplaçant vers la droite, elle transforme tous les 2 en 1 et, arrivée au premier 0, elle s'arrête. Il est facile de montrer que lorsque la machine prend l'état 2, le ruban ne contient plus de 1. Il n'y a donc pas d'ambiguïté quant à la définition de la machine pour l'état 2. On peut résumer la fonction de transition par le schéma suivant. [Ren01]

On montre l'exécution de la machine pour un ruban contenant deux 1.

1. État $0 \mapsto (2, \text{État } 1, \longrightarrow)$
   $\ldots 0\,0 \quad 0\,1\,1\,0\,0\,0 \quad 0\,0\ldots$
2. État $1 \mapsto (2, \text{État } 0, \longleftarrow)$
   $\ldots 0\,0 \quad 0\,1\,2\,0\,0\,0 \quad 0\,0\ldots$
3. État $0 \mapsto (2, \text{État } 0, \longleftarrow)$
   $\ldots 0\,0 \quad 0\,1\,2\,2\,0\,0 \quad 0\,0\ldots$
4. État $0 \mapsto (2, \text{État } 1, \longrightarrow)$
   $\ldots 0\,0 \quad 0\,1\,2\,2\,0\,0 \quad 0\,0\ldots$
5. État $1 \mapsto (2, \text{État } 1, \longrightarrow)$
   $\ldots 0\,0 \quad 0\,2\,2\,2\,0\,0 \quad 0\,0\ldots$

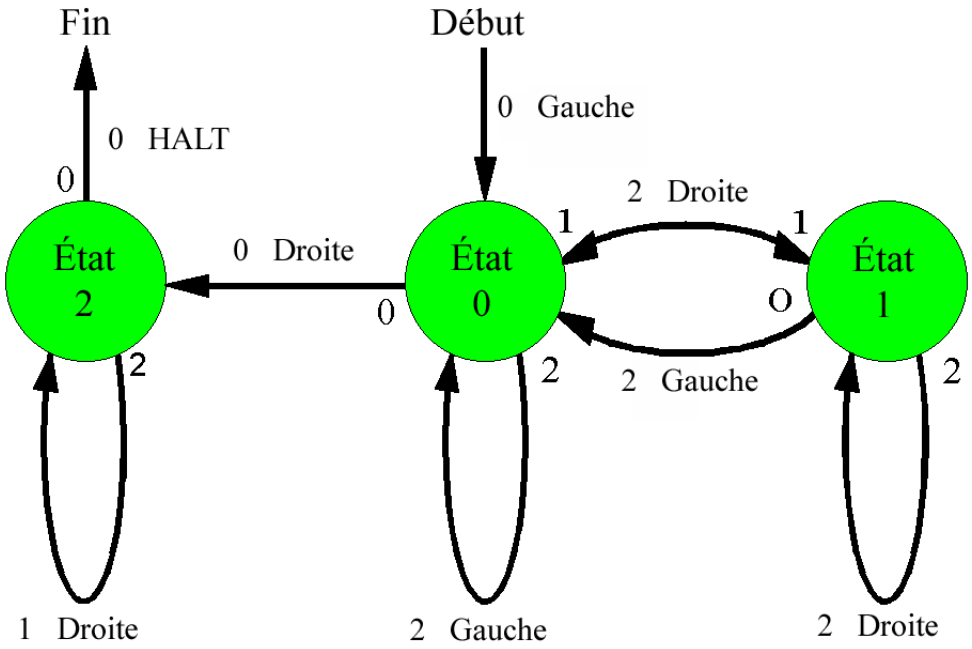


FIGURE 21.8 – Fonction de transition de la machine de TURING étudiée

6. État 1 $\mapsto$ (2, État 1, $\longrightarrow$)
   ...0 0 0 2 2 2 0 0 0 0 ...
7. État 1 $\mapsto$ (2, État 0, $\longleftarrow$)
   ...0 0 0 2 2 2 2 0 0 0 ...
8. État 0 $\mapsto$ (2, État 0, $\longleftarrow$)
   ...0 0 0 2 2 2 2 2 0 0 ...
9. État 0 $\mapsto$ (2, État 0, $\longleftarrow$)
   ...0 0 0 2 2 2 2 2 0 0 ...
10. État 0 $\mapsto$ (2, État 0, $\longleftarrow$)
    ...0 0 0 2 2 2 2 2 0 0 ...
11. État 0 $\mapsto$ (0, État 2, $\longrightarrow$)
    ...0 0 0 0 2 2 2 2 0 0 ...
12. État 2 $\mapsto$ (1, État 2, $\longrightarrow$)
    ...0 0 0 2 2 2 2 2 0 0 ...
13. État 2 $\mapsto$ (1, État 2, $\longrightarrow$)
    ...0 0 0 1 2 2 2 0 0 0 ...
14. État 2 $\mapsto$ (1, État 2, $\longrightarrow$)
    ...0 0 0 1 1 2 2 0 0 0 ...

15. État 2 $\mapsto$ (1, État 2, $\longrightarrow$)
    ...0 0 0 1 1 1 2 0 0 0...
16. État 2 $\mapsto$ (HALT)
    ...0 0 0 1 1 1 1 0 0 0...

Nous allons désormais décrire la machine de TURING de Paul RENDELL. [Ren11b]

21.3.1 Présentation générale

La machine que nous allons étudier est constituée de 4 parties :
— En vert : mémoire des états. Gère la fonction de transition.
— En jaune : le circuit gérant le changement d'état.
— En bleu : le ruban.
— En blanc : une structure remplissant des rôles plus complexes comme la lecture et l'écriture sur le ruban, qui sera décrite dans la suite.

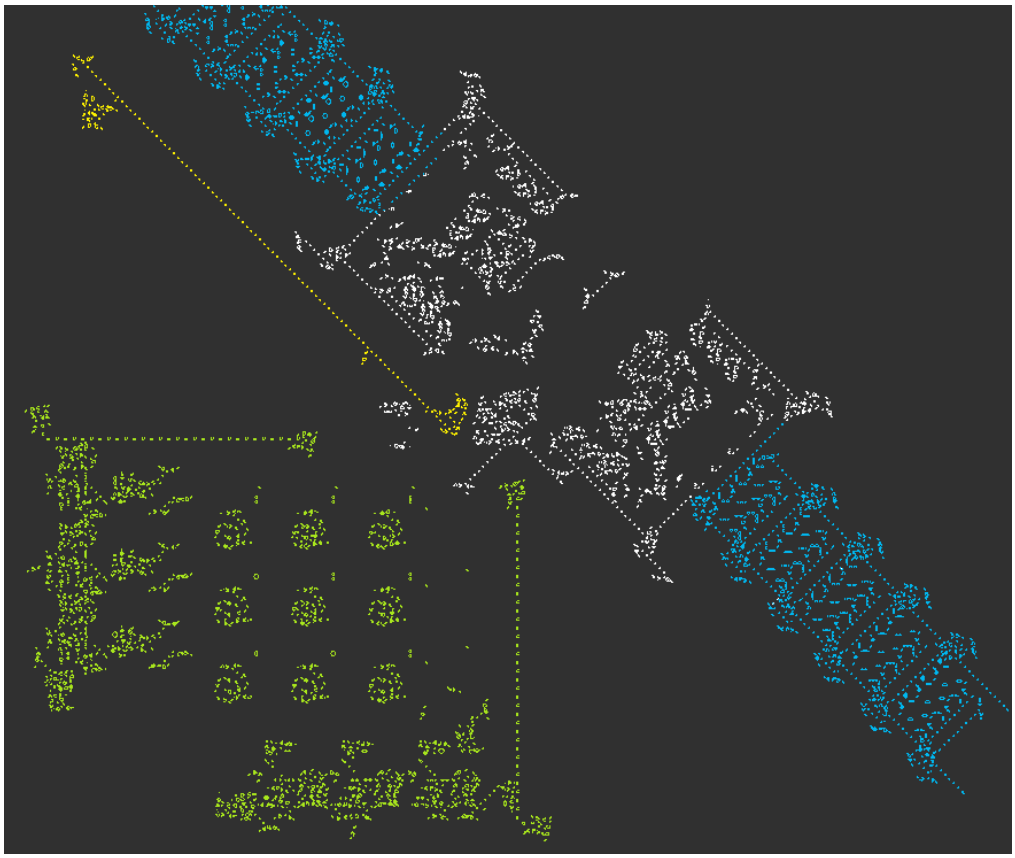


FIGURE 21.9 – Vue globale de la machine de TURING

21.3.2 La mémoire des états

Une cellule

La mémoire des états est composée de 3×3 cellules de mémoire. Chaque cellule correspond à un état de mémoire (0 à 2) et un état de la machine (0 à 2). La cellule contient des informations sur l'état suivant, le déplacement et ce qui doit être écrit sur le ruban sous forme de planeurs. Ils sont constamment dupliqués, mais un canon les empêche de sortir (la mémoire n'est pas destructive).

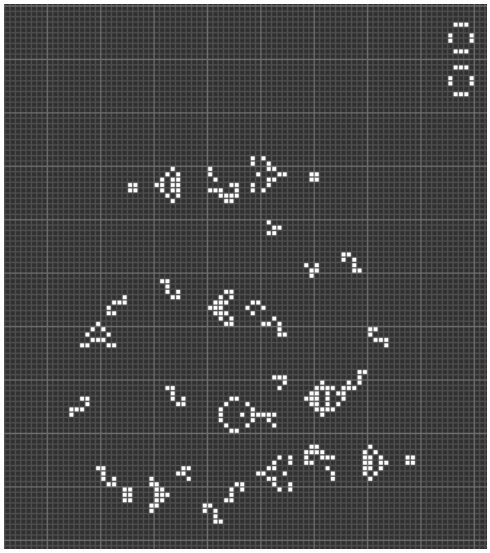


FIGURE 21.10 – Cellule de mémoire

Ainsi, pour que la cellule délivre son information, il faut tout d'abord couper le flux de planeurs qui sort du canon. Il sera rétabli une fois que l'information est sortie.

— En vert : Un flux de LWSS.
— En rouge : Le flux de planeurs sorti de la cellule.
— En orange : D'autres planeurs qui continuent de tourner dans la cellule.
— En blanc : Le reste de la cellule mémoire.

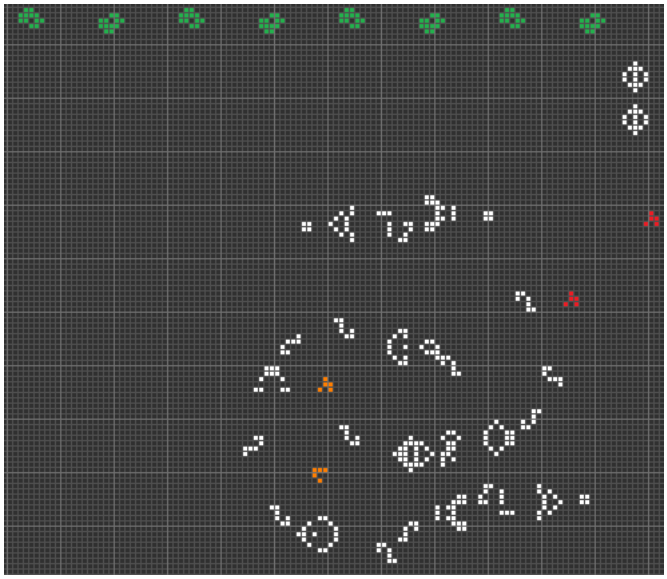


FIGURE 21.11 – Cellule de mémoire ayant émis son signal

De cette manière, certains vaisseaux du flux de LWSS sont détruits (ils correspondent aux planeurs émis précédemment). On voit que le canon de la case mémoire (en bleu) émet de nouveau des planeurs.

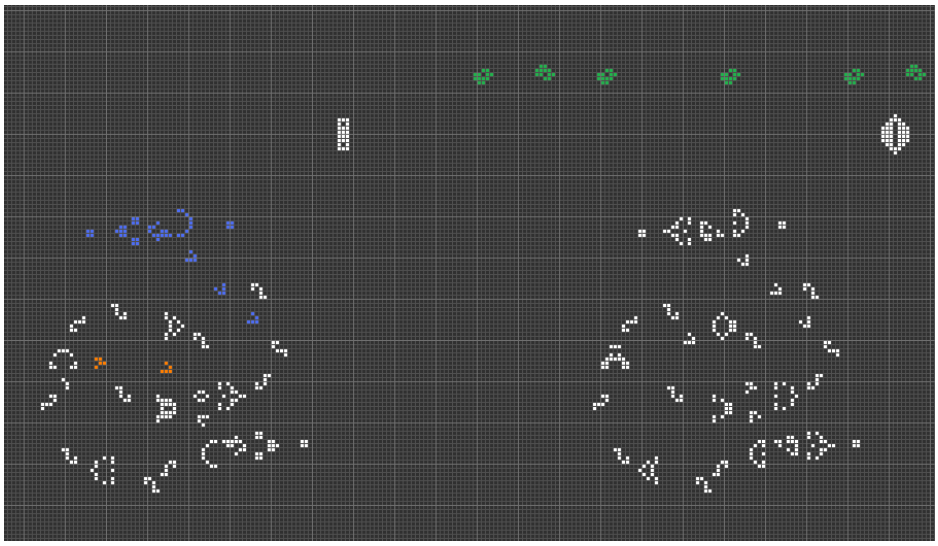


FIGURE 21.12 – Flux de LWSS filtré

Enfin, on prend le complémentaire du flux de LWSS (en bleu sur la figure).

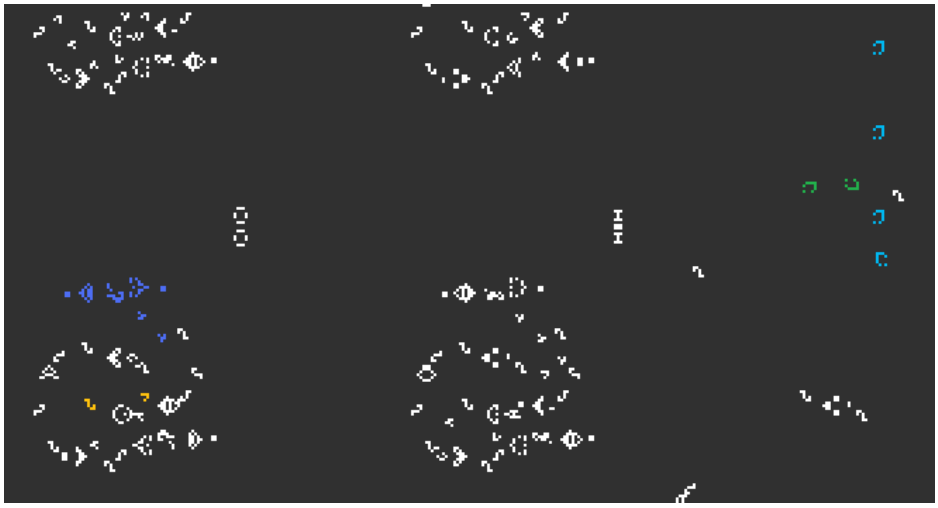


FIGURE 21.13 – Calcul du complémentaire

Ce flux entre va ensuite être traité par la machine.

Signification des LWSS émis

8 LWSS sortent de la zone de transition. Chacun correspond à un bit. Voici leur description sur la machine en question.

Bit	Signification
1	Passage dans l'état 0
2	Écrire 1
3	Écrire 2
4	Inutilisé
5	Passage dans l'état 1
6	Passage dans l'état 2
7	Inutilisé
8	Inutilisé

TABLE 21.1 – Description du codage des transitions

Si aucun des bits 2 et 3 n'est présent, un '0' est écrit sur le ruban.

21.3.3 Lecture du ruban

Nous allons suivre le trajet d'un planeur, depuis sa sortie de la mémoire des états jusqu'à l'arrivée de l'information contenue sur le ruban, sous forme de

planeurs, dans la mémoire des états finis.

Le planeur orange va déclencher le processus de lecture. Il est dupliqué par la structure en rose, puis par celle en vert, après avoir été réfléchi deux fois par les réflecteurs bleus.

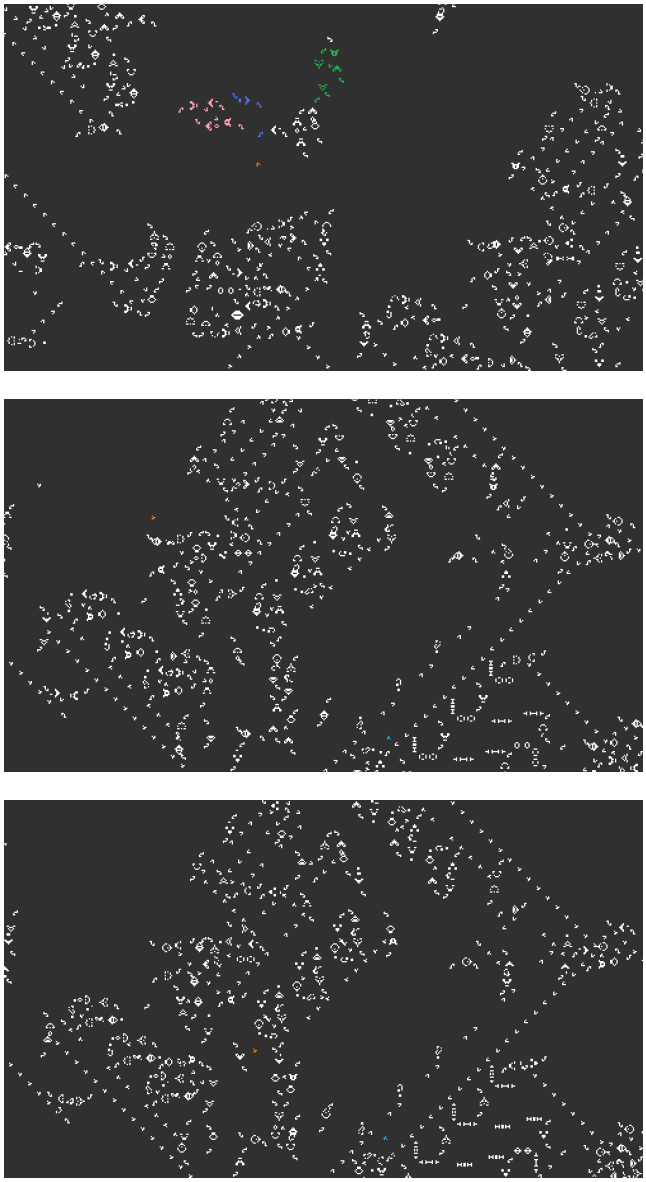


FIGURE 21.14 – Déclenchement du processus de lecture

Ensuite, ce planeur (toujours orange) est dupliqué par le réflecteur en rouge sur l'image. Seul celui qui se dirige vers le bas nous intéresse pour la lecture du ruban.

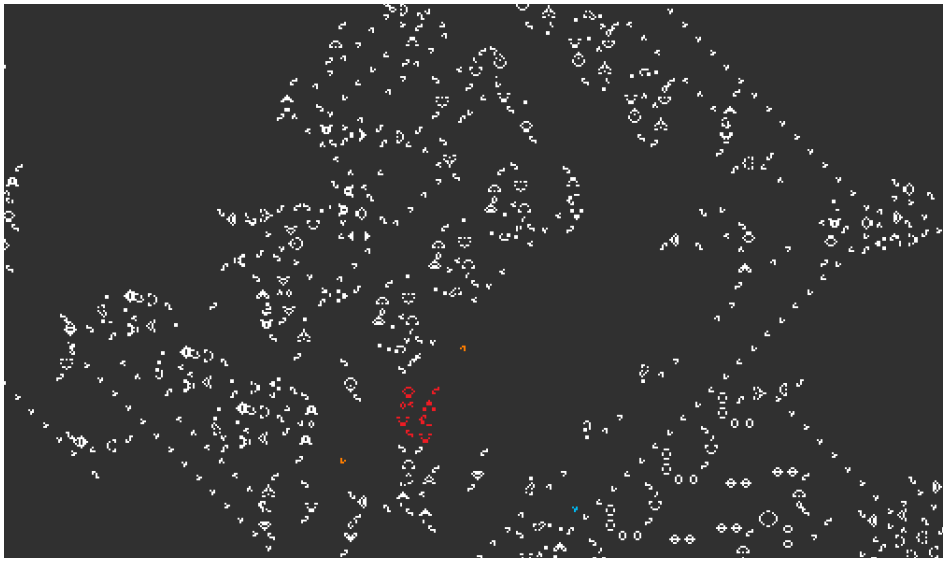


FIGURE 21.15 – Duplication du planeur déclencheur

On voit sur cette image où est stockée la mémoire. Un planeur (en bleu clair) "rebondit" entre les 2 flots de planeurs, il contient la valeur de la case du ruban (un planeur code pour la valeur 1). Lors de ses rebonds, il marque le flux de planeurs orange. Le planeur orange va faire un trou dans le flux en orange, de manière à ce que le planeur en jaune sur l'image puisse être libéré.

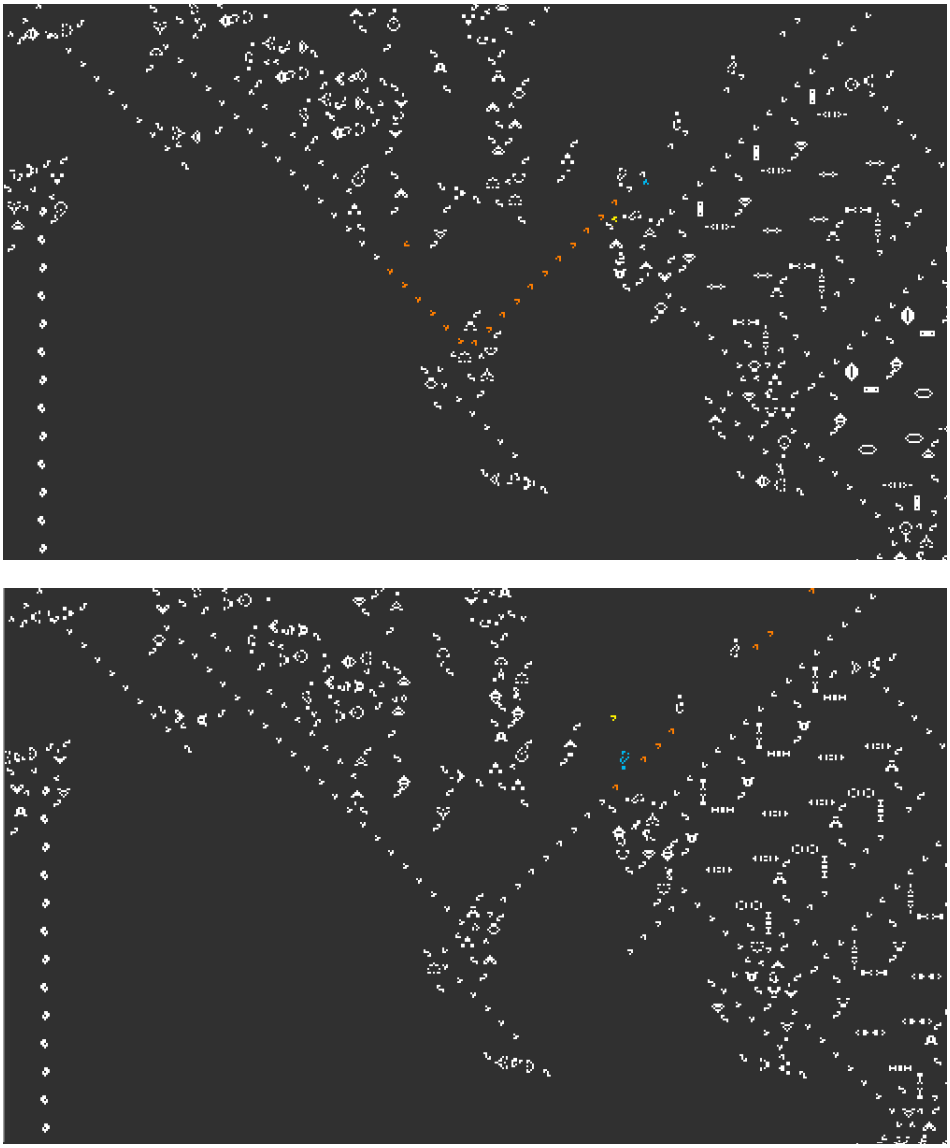


FIGURE 21.16 – Extraction de la mémoire du ruban

Avant que le planeur jaune n'atteigne la structure verte, le canon bloque le flux de planeurs bleus qui contient l'information sur le contenu de la case. Le planeur jaune bloque le flux vert, donc autorise les 2 planeurs bleus à sortir. Ils sont ensuite dirigés directement dans la mémoire des états.

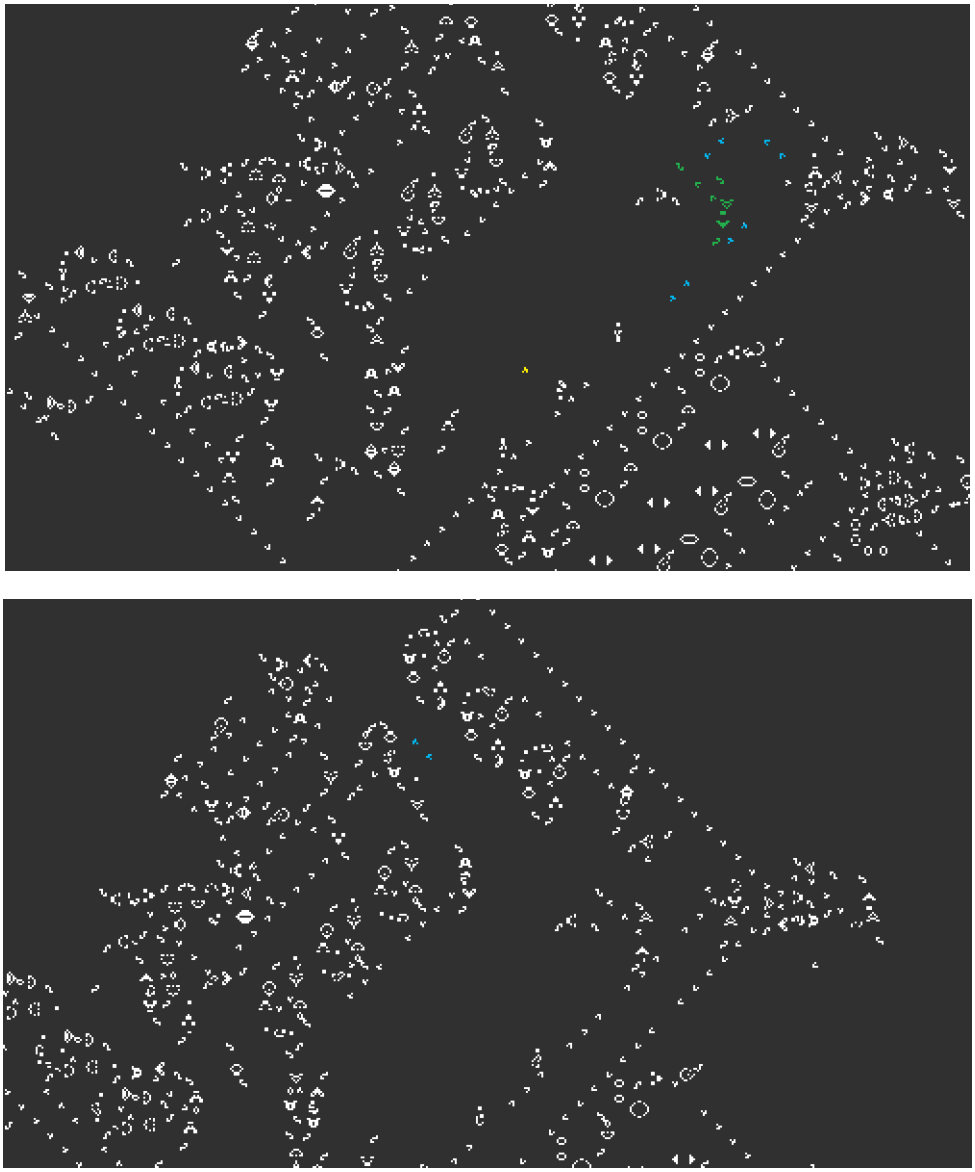


FIGURE 21.17 – Redirection de la mémoire lue

21.3.4 Décalage du ruban

Le ruban peut être décalé, suivant la fonction de transition, vers la droite ou vers la gauche. Pour des raisons de simplicité, la machine ne déplace pas sa tête de lecture, mais décale le contenu du ruban dans la direction opposé. Ce décalage n'est pas immédiat. Il y a en effet un temps de propagation de l'information. Aussi, les zones éloignées du ruban pourront ne pas avoir commencé leur décalage alors que la transition est déjà terminée. Mais ceci n'est pas problé-

matique car cette information distante ne peux pas être utilisée avant un temps supérieurs aux temps de propagation. De plus, les systèmes de synchronisation empêchent les collisions entre deux cases successives du ruban.

On se ramène ici à l'étude des deux décalages sur la partie droite du ruban. Nous décrirons les 2 décalages possibles.

Vers la gauche

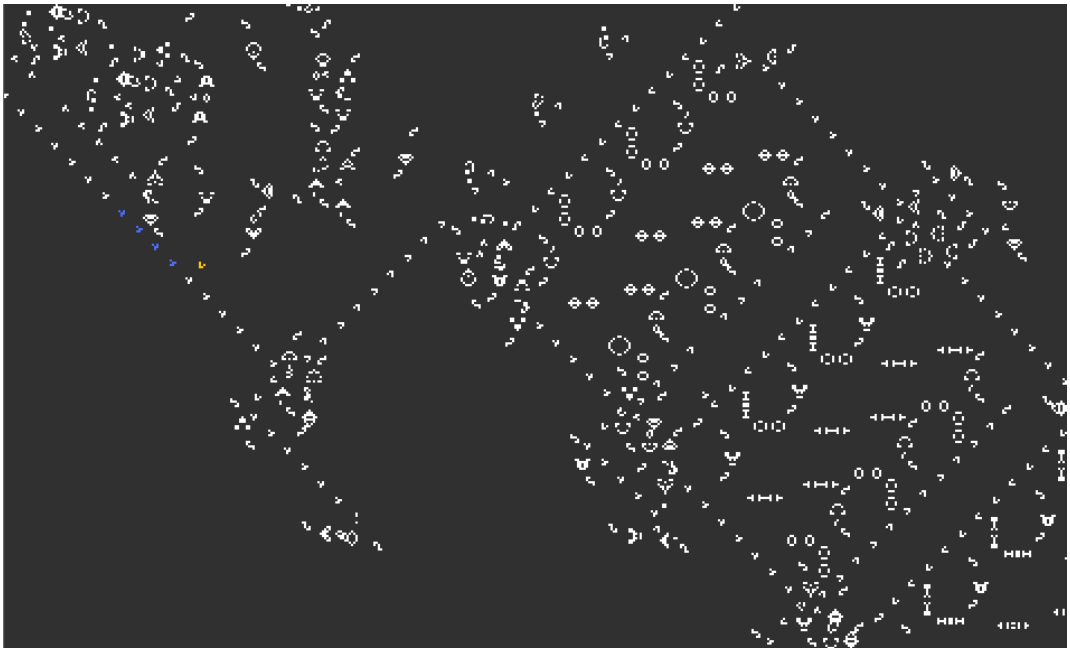


FIGURE 21.18 – Première étape du décalage vers la gauche

On reprend la figure 21.16. Le même planeur va déclencher le décalage. Les 4 planeurs bleus sont détruits.

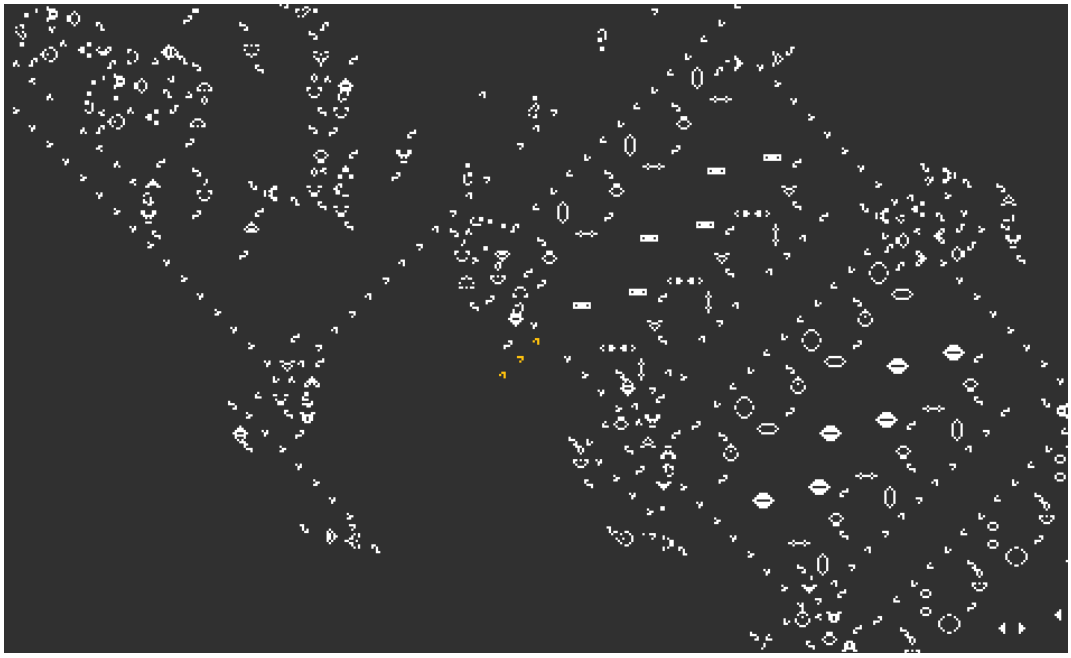


FIGURE 21.19 – Seconde étape du décalage vers la gauche

Le complémentaire du flux se retrouve ici, coloré en jaune.

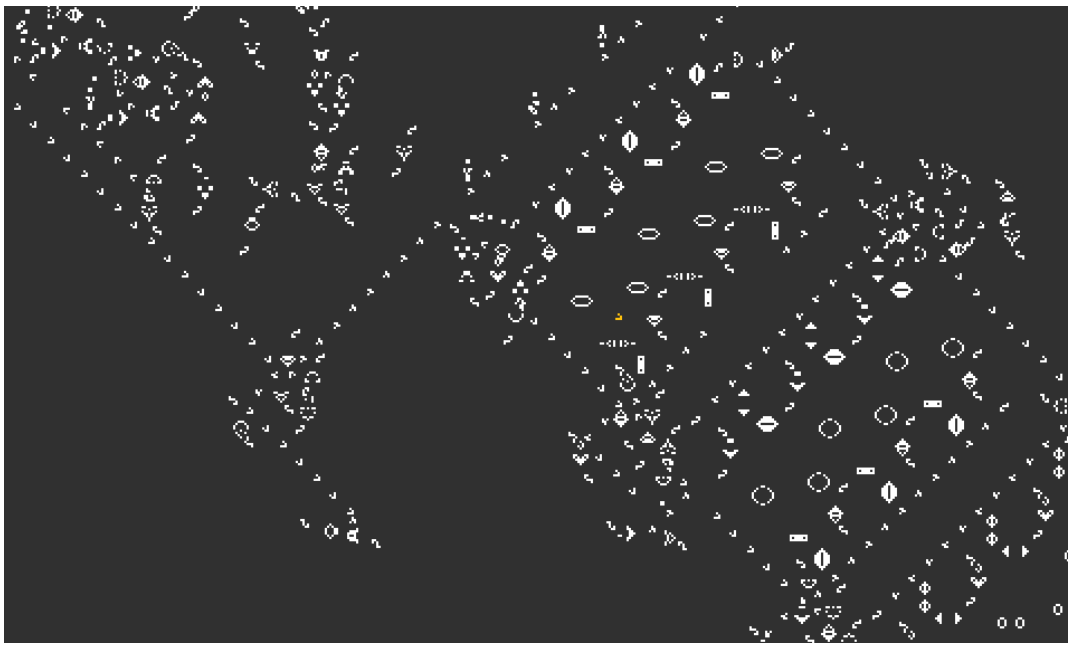


FIGURE 21.20 – Dernière étape du décalage vers la gauche

Chaque compartiment de mémoire laisse remonter son planeur dans le compartiment au dessus.

Vers la droite

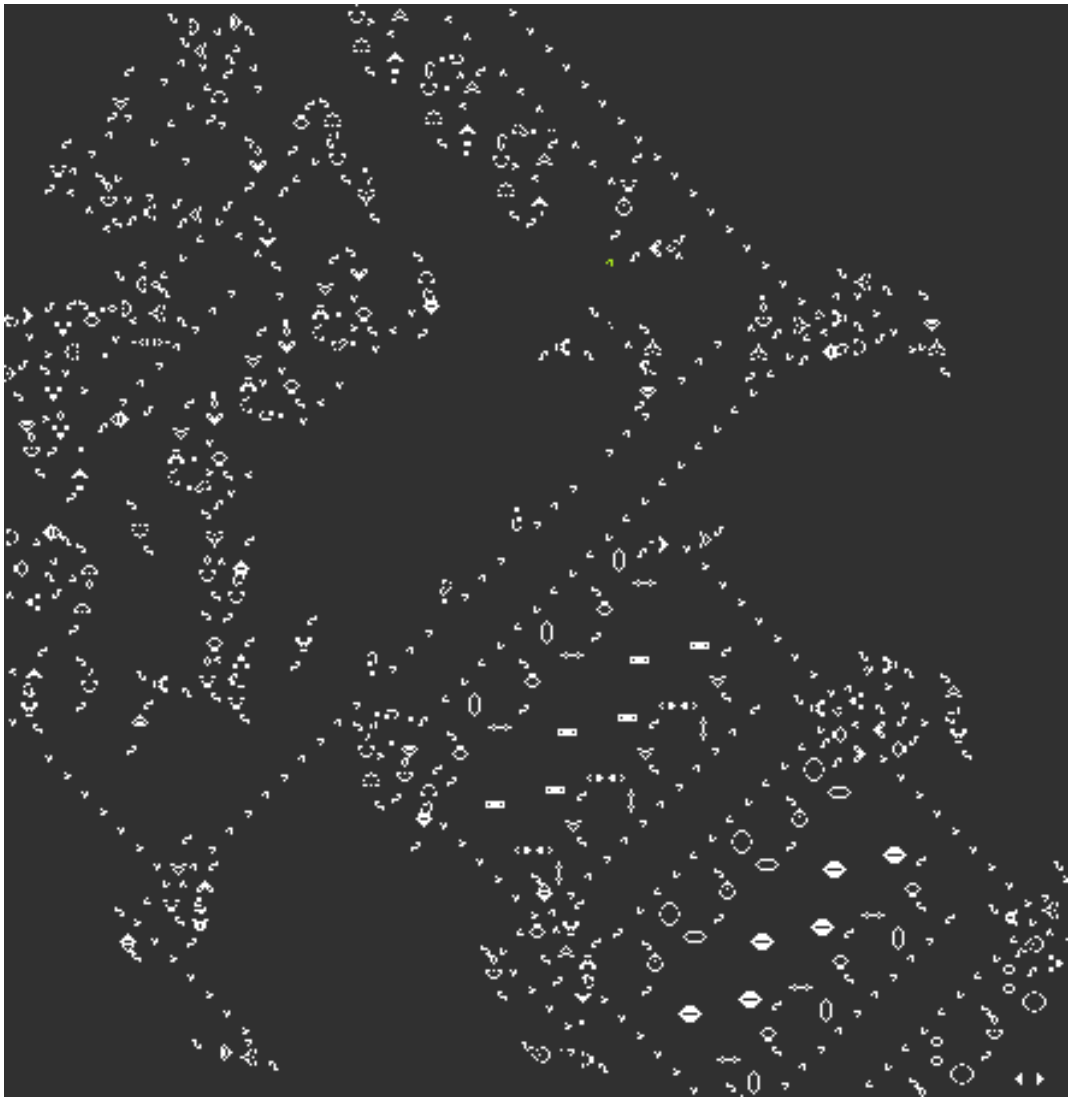


FIGURE 21.21 – Le planeur vert sur la figure va déclencher le décalage vers la droite du ruban. Il détruit 4 planeurs du flux où il se dirige.

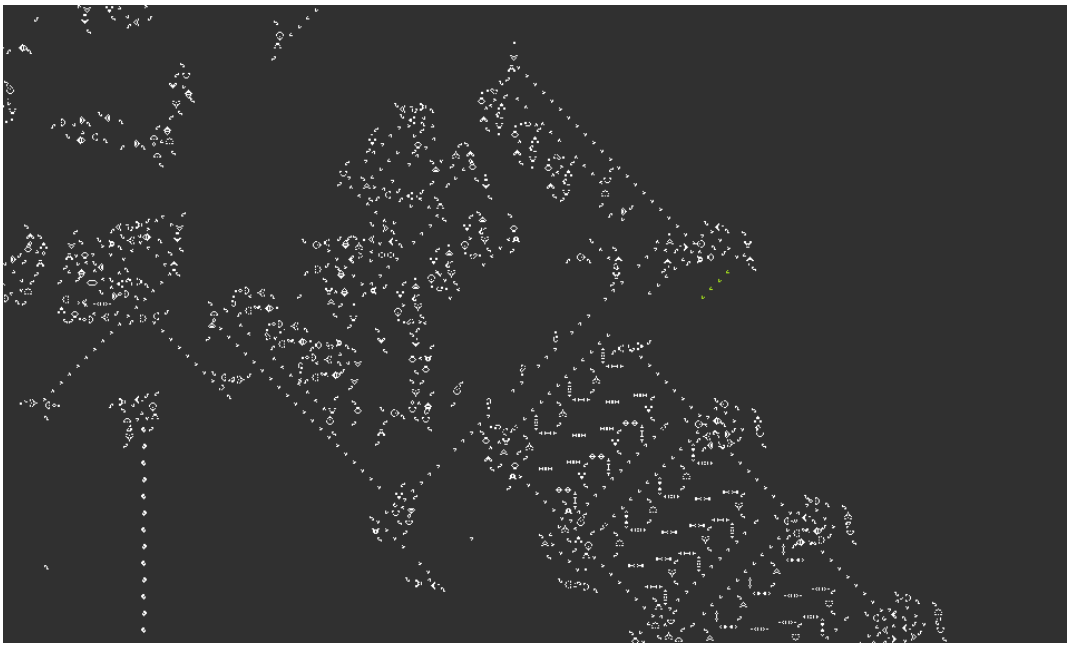


FIGURE 21.22 – Par un jeu de réflecteurs (et une porte NON), un flux est émis vers le ruban, colorié ici en vert.

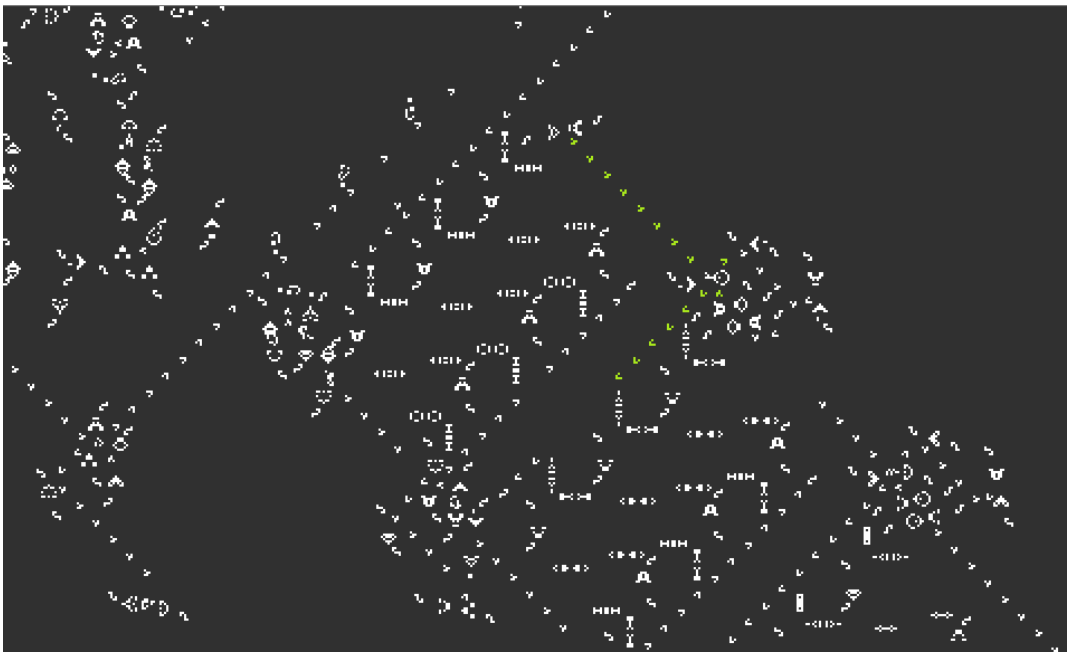


FIGURE 21.23 – Ce flux va provoquer une coupure dans le ruban, par lequel les planeurs de mémoire passent vers la case voisine.

21.3.5 Gestion de l'état suivant

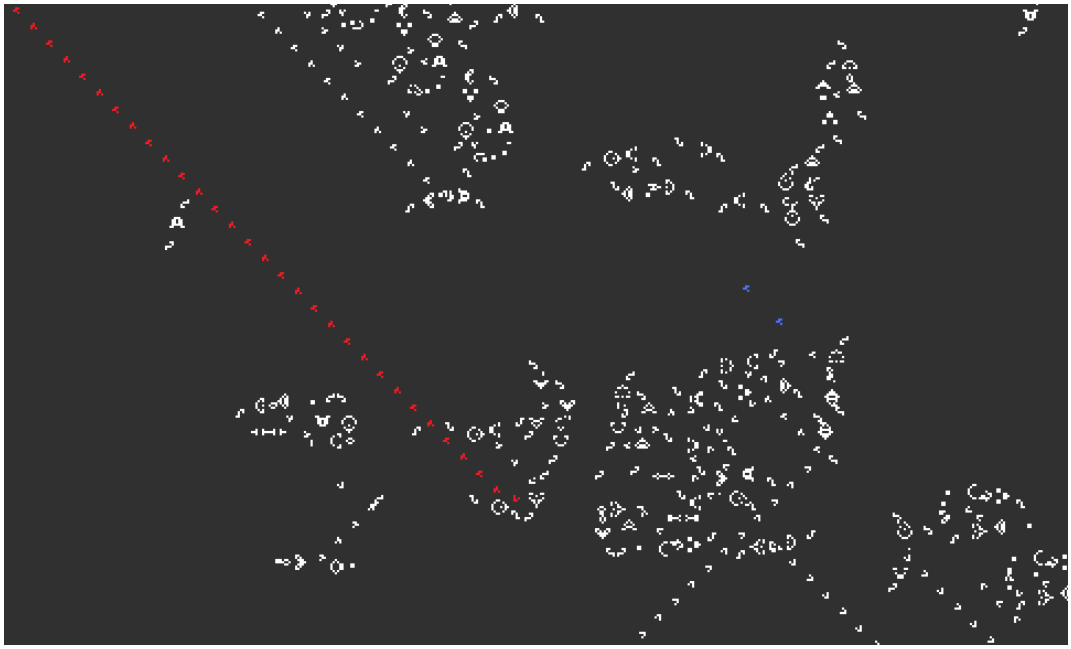


FIGURE 21.24 – L'information sortant de la mémoire des états contient la valeur du prochain état. Ce sont les deux planeurs bleus ici. Le flux rouge sera marqué par ces deux planeurs bleus.

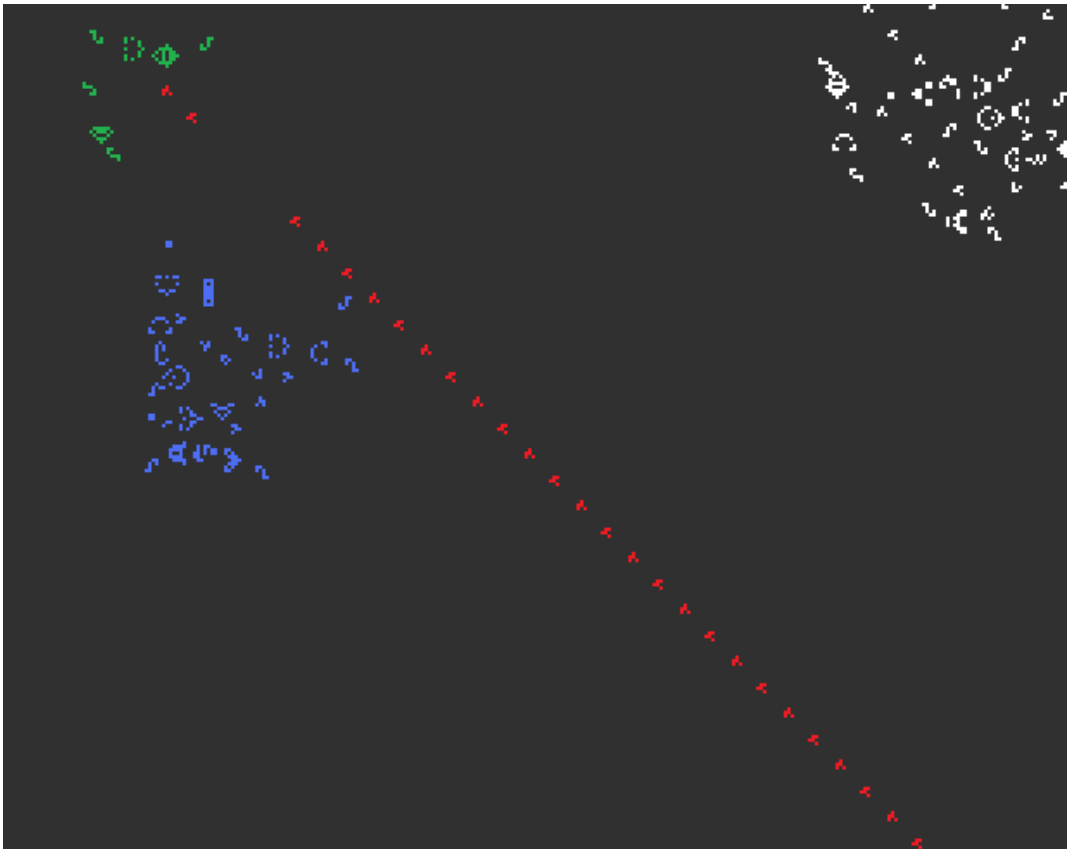


FIGURE 21.25 – Le complémentaire de ce flux est généré par la figure verte.

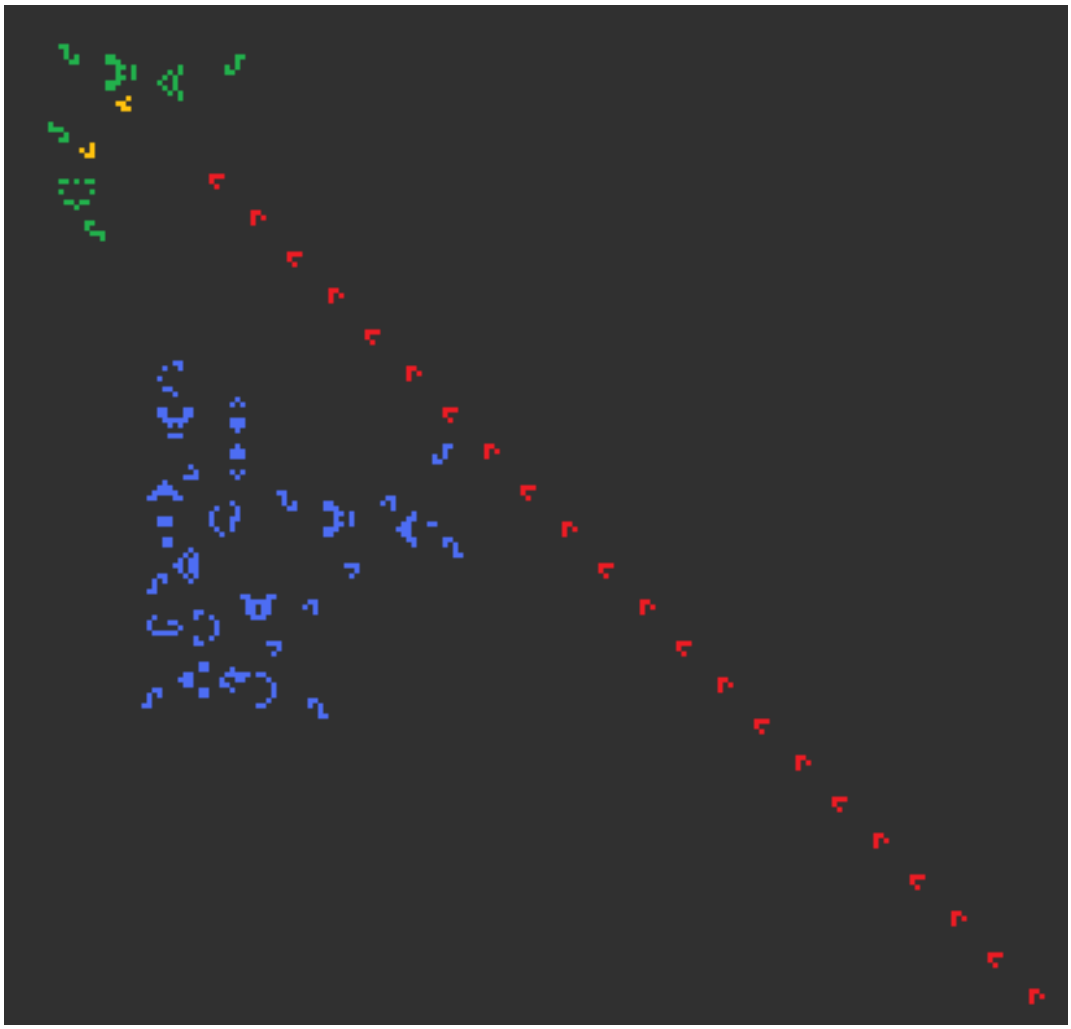


FIGURE 21.26 – On voit ici les planeurs en jaune causés par le trou dans le flux rouge.

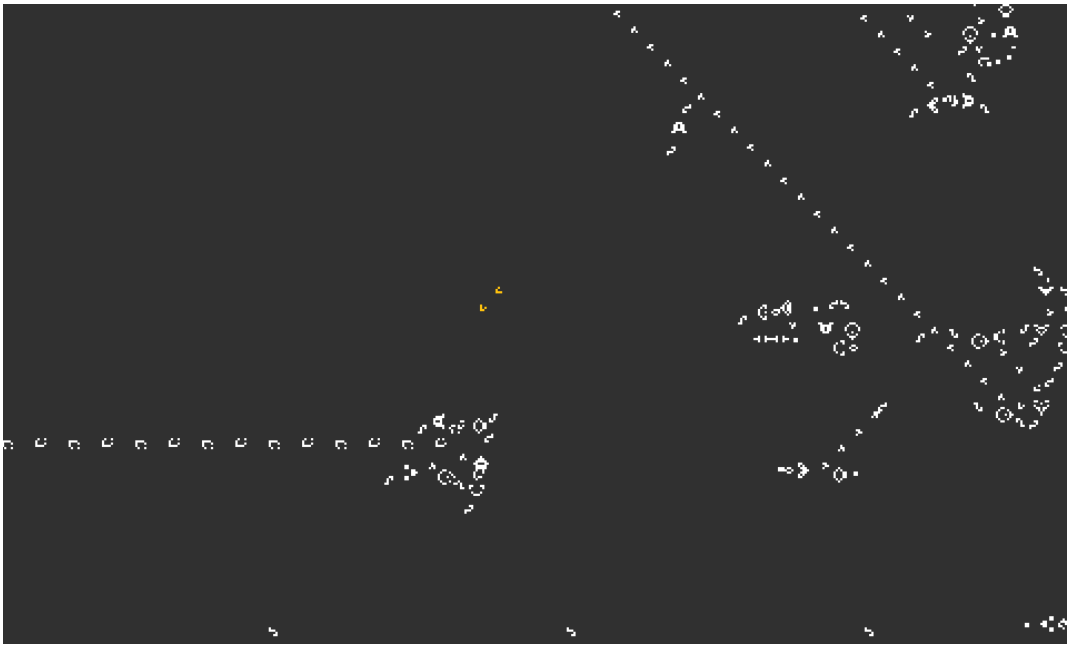


FIGURE 21.27 – Enfin, ces deux planeurs arrivent dans la mémoire des états finis, où la fonction de transition sera appelée.

21.4 Conception d'une machine de TURING universelle

La machine précédente ne peut supporter que trois états, et un alphabet de trois valeurs (0, 1 ou 2). Cependant, elle peut être généralisée avec un nombre plus important d'états et un alphabet plus grand. Deux problèmes apparaissent pour la construction d'une machine de TURING :

— Le nombre de cases du ruban est fini. La machine que nous allons présenter possède une mémoire qui se construit au fur et à mesure, de telle sorte qu'on ne puisse arriver au bout, puisque la vitesse de construction est supérieure à la vitesse de déplacement. La mémoire peut ainsi être considérée comme infinie.

— Cette machine doit pouvoir exécuter n'importe quelle machine codée sur son ruban. On sait qu'il existe des machines de TURING universelles, que l'on peut coder à partir d'un nombre fini d'états et de valeurs de mémoire. La machine que nous présentons simule donc l'exécution d'une autre machine de TURING.

On appelle $\mathcal{U}$ la machine de TURING universelle et $\mathcal{M}$ la machine simulée par $\mathcal{U}$. Dans un premier temps, $\mathcal{U}$ stocke sur le ruban la fonction de transition de $\mathcal{M}$. $\mathcal{U}$ réserve une partie de son ruban à la description de $\mathcal{M}$, une partie à la description de la configuration en cours de $\mathcal{M}$ et utilise le reste de son ruban pour stocker le ruban de $\mathcal{M}$.

21.4.1 Une machine de TURING universelle dans le jeu de la vie

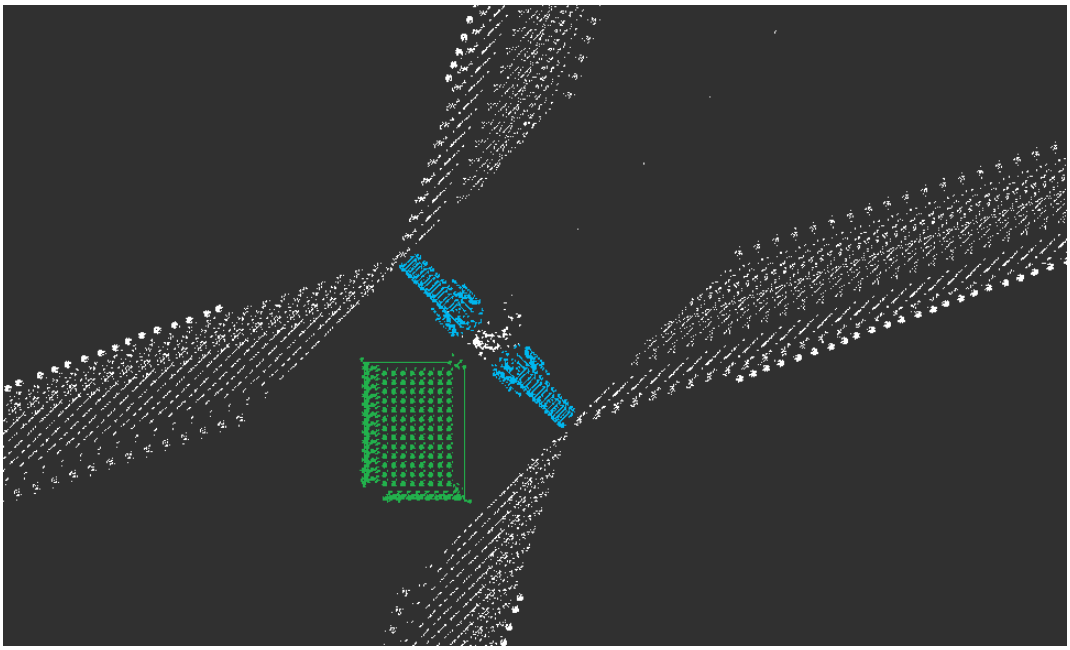


FIGURE 21.28 – Vue globale de la machine

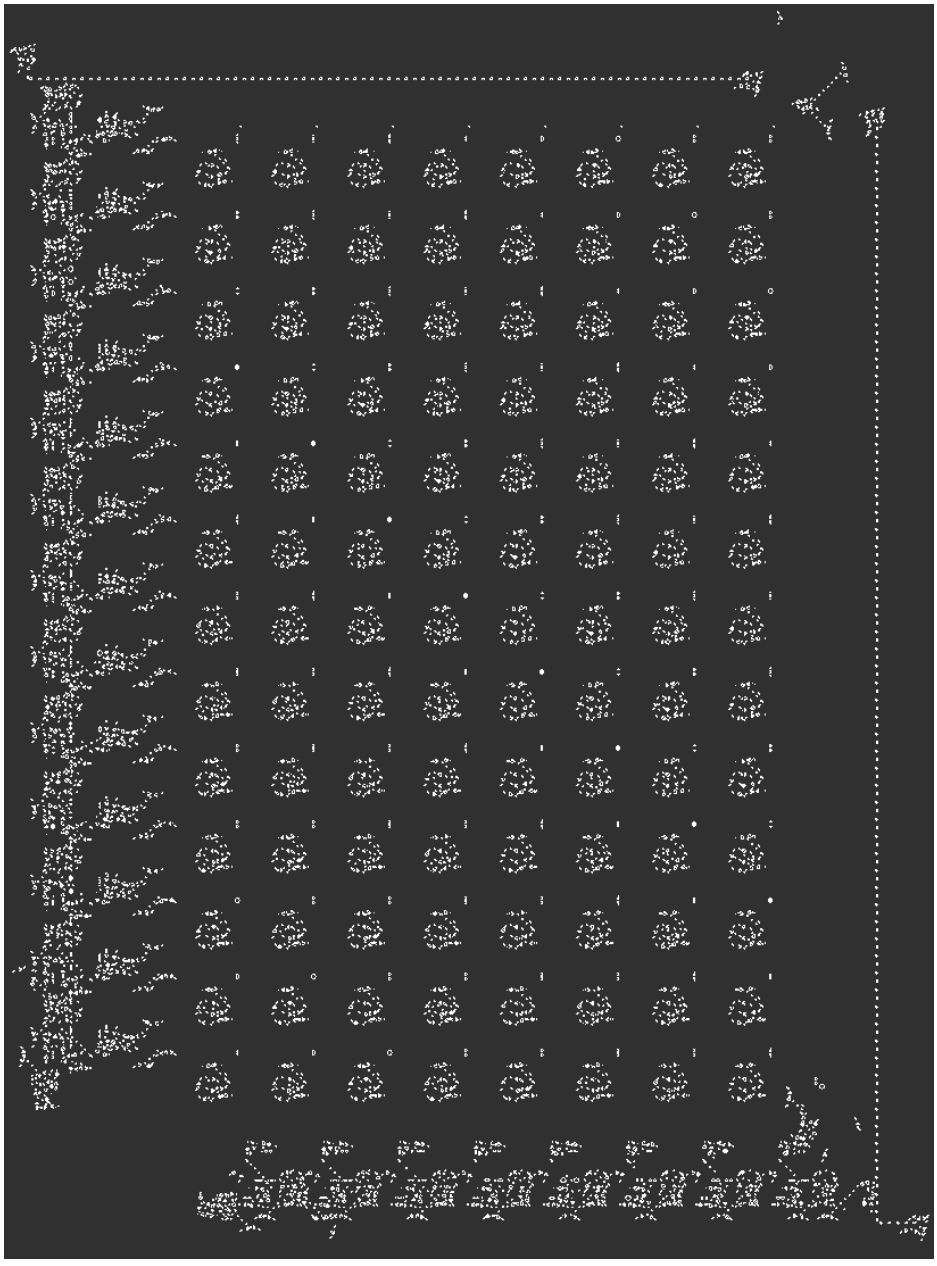


FIGURE 21.29 – La mémoire des états finis (13 états $\times$ 8 symboles)

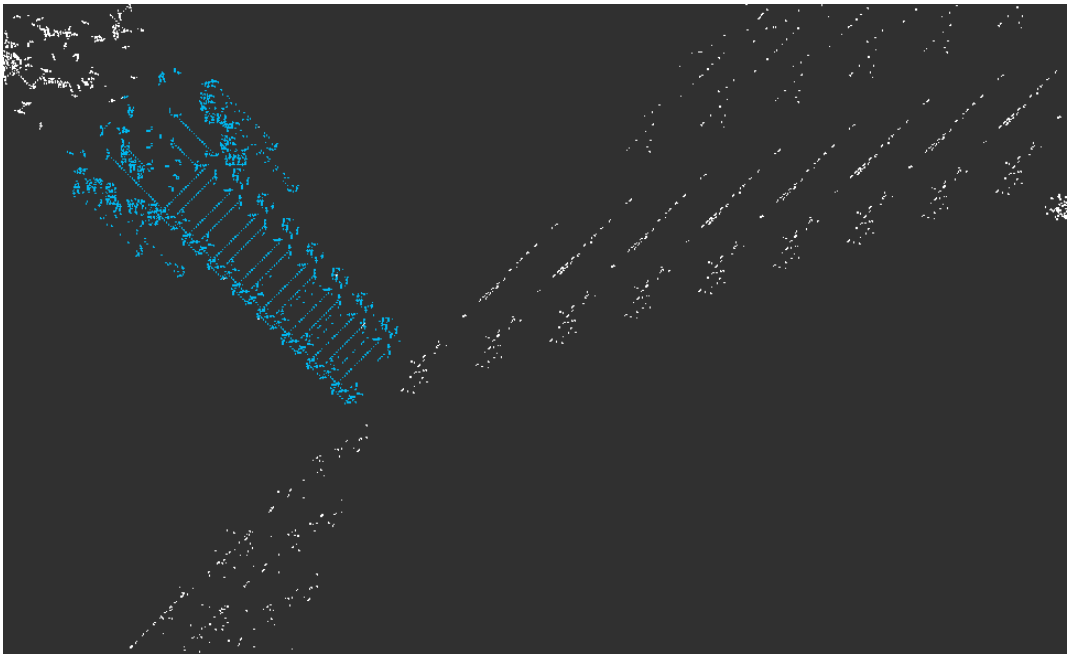


FIGURE 21.30 – La mémoire est assemblée ici (en bleu la partie déjà créée du ruban).

21.5 Simulation des automates cellulaires grâce à un langage TURING-équivalent

Pour prouver que les automates cellulaires ne sont pas plus puissants que les machines de TURING, on pourrait écrire un code dans un langage informatique qui serait capable de simuler tous les automates cellulaires dont on donnerait la description en entrée. Cependant on s'épargnera cette peine en utilisant des programmes déjà existant comme Golly [RTH+12].

La seconde partie de l'argumentation s'appuie sur le fait qu'un ordinateur ne peut exécuter que des instructions en assembleur, comparables aux instructions d'une machine RAM. Chacune de ces instructions pouvant être simulée par une machine de TURING, les ordinateurs sont au plus aussi puissants qu'une machine de TURING. Cet argument, malgré son apparence informelle, va être utilisé plusieurs fois et est tout à fait valable mais nécessiterait, dans l'idéal, de prouver que chaque instruction ou opération élémentaire peut être simulée par une machine de TURING.

21.6 Conclusion

Théorème 20 (TURING-équivalence des automates cellulaires).

La classe des fonctions calculées par les automates cellulaires est égale à la classe des fonctions calculées par les machines de TURING.

Démonstration. Cela équivaut à dire qu'une fonction est calculable au sens des automates cellulaires si, et seulement si, elle est calculable au sens des machines de TURING.

($\Leftarrow$) : Soit f une fonction calculable au sens des machines de TURING et $\mathcal{U}$ la machine de TURING universelle exhibée dans le jeu de la vie de CONWAY. Il existe une machine de TURING $\mathcal{M}$ qui calcule f. En simulant $\mathcal{M}$ dans $\mathcal{U}$ et en simulant l'exécution de $\mathcal{U}$ dans le jeu de la vie, on obtient que f est calculable au sens des automates cellulaires.

($\Rightarrow$) : Soit $\mathcal{C}$ une configuration finie d'un automate cellulaire $\mathfrak{A}$. Le voisinage de $\mathfrak{A}$ étant fini, la configuration suivant $\mathcal{C}$, l'est aussi. On peut donc calculer la transition à l'aide d'un ordinateur, donc d'une machine de TURING. Ainsi, toute configuration finie de $\mathfrak{A}$ est calculable au sens des machines de TURING.

On a donc bien l'équivalence entre les deux modèles de calcul.

□

Chapitre 22

Les machines de TURING non déterministes

22.1 Définition

Définition 77 (Machine de TURING non deterministe)**.**

Une machine de TURING non déterministe (NDTM) est un 7-uplet $(Q, \Gamma, B, \Sigma, q_0, \delta, F)$ où

- Q est un ensemble fini d'états,
- Γ est l'alphabet de travail,
- $B \in \Gamma$ est un symbole particulier dit symbole blanc,
- Σ est l'alphabet des symboles en entrée ($\Sigma \subseteq \Gamma \setminus \{B\}$)
- $q_0 \in Q$ est l'état initial,
- $\delta : Q \times \Gamma \to \mathcal{P}\left(Q \times \Gamma \times \{\longleftarrow, \longrightarrow, \texttt{HALT}\}\right)$ est la fonction de transition,
- $F \subseteq Q$ est l'ensemble des états acceptants.

Ce modèles est, à s'y méprendre, proche des machines de TURING. La différence réside dans le fait que la fonction de transition décrit plusieurs transitions possible pour un même couple (q, a).

Un calcul est toujours une suite valide de configurations valide finissant quand le signal `HALT` est reçu. Un calcul est acceptant s'il termine dans un état acceptant. Une machine de TURING non déterministe a accepte le mot en entré s'il existe un calcul acceptant.

On voit par conséquent que ces machines sont faites pour la décision et non pour le calcul. En effet, il est évident que le ruban est généralement différent

pour chaque calcul, même pour des calculs acceptants.

D'une façon plus automatique, on peut imaginer que la machine se réplique à chaque étape en autant de nouvelles machines qu'il y a de transitions possibles. On obtient alors un arbre de configuration dont la racine est la configuration initiale. Et chaque arrête est une transition valide. L'arbre peut être de profondeur infinie. Une feuille est acceptante si elle décrit une configuration acceptante. Un nœud est acceptant si au moins un de ses descendant est acceptant. Enfin, la machine accepte son entrée si la racine est acceptante.

22.2 TURING-équivalence

Proposition 83.

Les machines de TURING non déterministes sont plus puissantes que les machines de TURING.

Démonstration. En effet, ces machines ne sont qu'une restrictions des machines de TURING non déterministes. □

Proposition 84.

Les machines de TURING sont plus puissantes que les machines de TURING non déterministes.

Démonstration. Pour faire cette preuve, on va chercher s'il existe un calcul acceptant d'une machine de TURING non déterministe donnée.

On se donne $\mathcal{N}$ une machine de TURING non déterministe et une entrée x sur l'alphabet d'entré de $\mathcal{N}$.

Pour trouver un calcul acceptant, on cherche une feuille dans l'arbre de configuration précédemment décrit. Si il existe un calcul qui ne termine, cet arbre n'est pas de profondeur finie. Il reste toutefois à branchement fini. Pour trouver une feuille, il suffit de faire un parcours en largeur. Il y a alors équivalence entre la visite d'une feuille en un temps fini et l'existence d'au moins une feuille.

Cependant, ce parcours n'est pas simple puisque le ruban de chaque configuration peut différer. Il est alors nécessaire de se souvenir du ruban.

On utilise pour cela une machine de TURING $\mathcal{D}$ à trois rubans. Le premier ruban contient et conserve l'entrée de $\mathcal{N}$, le second contient l'exécution courante de $\mathcal{N}$ et le troisième le point dans le parcours de l'arbre de configuration.

Lors du parcours, on ne retient pas le ruban à chaque état mais on retient la liste des choix non déterministes faits. On retrouve le ruban courant car on conserve le ruban initial. Il suffit alors d'implémenter un parcours en largeur avec une file stockée sur le troisième ruban. Chaque nœud est identifié par la liste des choix non déterministes. Aussi, pour ajouter les fils d'un nœud à la file, il suffit d'y ajouter la description du nœud courant auquel on ajoute les le choix non déterministe qui y mène. La machine parcours ainsi l'arbre de configuration en largeur et trouve une feuille si et seulement si il en existe une.

Ainsi, l'arrêt et l'acceptation d'une machine de TURING non déterministe est décidable par machine de TURING. Par conséquent, les machines de TURING non déterministes sont moins puissantes que les machines de TURING. □

Théorème 21.

Les machines de TURING non déterministes et les machines de TURING sont équivalentes.

Démonstration. Les deux propositions précédentes permettent d'établir ce résultat. □

Cinquième partie

Des modèles de calcul moins puissants

Chapitre 23

Les automates finis

On va d'abord présenter un modèle de calcul très limité. Il se caractérise par un fonctionnement proche d'une machine de TURING mais sans accès à un autre type de mémoire que son état interne.

23.1 Les automates finis non déterministes

Définition 78 (Automate fini non déterministe)**.**

Un automate fini non déterministe est un 5-uplet $(Q, \Sigma, q_0, \delta, F)$ où
- Q est un ensemble fini d'états,
- Σ est l'alphabet de l'automate,
- $q_0 \in Q$ est l'état initial,
- $\delta \subseteq Q \times \Sigma \times Q$ est l'ensemble des transitions,
- $F \subseteq Q$ est l'ensemble des états acceptant.

Pour tout 3-uplet $(q, a, q') \in \delta$, on note $(q, a) \to q'$.
Soit un automate fini $A = (Q, \Sigma, q_0, \delta, F)$
Soit $u \in \Sigma^\star$. On note $u = u_0 \dots u_{n-1}$.
Une configuration de l'automate fini est de la forme $(q_i, u_i \dots u_{n-1})$.
Une transition correspond au passage de l'automate d'un état dans un autre et on note $(q, u) \to (q', v)$. L'automate peut effectuer cette transition si $\exists a \in \Sigma, av = u \wedge (q, a) \to q'$. La lecture d'une lettre par l'automate change son état.
Un calcul de u sur A est une suite de configuration qu'on peut réaliser par une suite de transition et dont la configuration initiale est (q_0, u).

$$(q_0, u_0 \dots u_{n-1}) \to (q_1, u_1 \dots u_{n-1}) \to \dots \to (q_n, \varepsilon)$$

On dit que le calcul est acceptant si $q_n \in F$. Le mot u est reconnu par A s'il existe un calcul acceptant. L'ensemble des mots reconnus par A est appelé le langage de A et est noté $L(A)$.

Notation 32.

L'ensemble des langages reconnus par automate fini est noté **AF**.

23.2 Les automates asynchrones

Définition 79 (Automate asynchrone).

Un automate asynchrone est un 5-uplet $(Q, \Sigma, q_0, \delta, F)$ où
- Q est un ensemble fini d'états,
- Σ est l'alphabet de l'automate,
- $q_0 \in Q$ est l'état initial,
- $\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$ est l'ensemble des transitions,
- $F \subseteq Q$ est l'ensemble des états acceptant.

Un calcul sur un automate asynchrone est légèrement différent. En effet, une transition $(q, u) \to (q', v)$ est valide si $(\exists a \in \Sigma, av = u \wedge (q, a) \to q') \vee (u = v \wedge (q, \varepsilon) \to q')$. L'automate peut donc changer d'état sans lire de lettre du mot en entrée. On appelle ce genre de transition une ε-transition. Ainsi, la longueur du calcul est variable.

Les autres définitions sont les mêmes que celles concernant l'automate fini non déterministe.

Théorème 22 (Théorème de synchronisation).

Les langages reconnus par les automates asynchrones sont exactement ceux reconnus par les automates finis non déterministes.

Démonstration. Soit A un automate asynchrone. On construit un automate N un automate fini non déterministe.

On fixe $Q(N) = Q(A)$, $\Sigma(N) = \Sigma(A)$, $q_0(N) = q_0(A)$.

On construit ensuite l'automate N en éliminant les ε-transitions par un algorithme de fermeture transitive.

On commence par initialiser l'ensemble des états finaux de N à $F(A)$.

Pour tout $(s, t, u) \in Q^3$ tel qu'il existe un chemin d'ε-transition entre s et t et une transition étiqueté par $a \in \Sigma$ dans A, alors, on ajoute la transition $(s, a) \to u$ dans N.

Pour chaque chemin d'un état s à un état t terminal formé de ε-transitions, on ajoute s à l'ensemble des états terminaux de N.

Ainsi, tout chemin dans A trouve un équivalent dans N avec les mêmes états de départ et d'arrivé.

Réciproquement, tout automate fini non déterministe est un automate asynchrone. □

23.3 Les automates finis déterministes

Définition 80 (Automate fini déterministe).

Un automate fini déterministe est un 5-uplet $(Q, \Sigma, q_0, \delta, F)$ où
- Q est un ensemble fini d'états,
- Σ est l'alphabet de l'automate,
- $q_0 \in Q$ est l'état initial,
- $\delta \subseteq Q \times \Sigma \times Q$ est l'ensemble des transitions, tel que $\forall (q, a) \in Q \times \Sigma, |\{q' \in Q \mid (q, a) \to q'\}| \leqslant 1$,
- $F \subseteq Q$ est l'ensemble des états acceptant.

Cela revient à voir δ comme une fonction partielle de $Q \times \Sigma \to Q$.

Dans ce cas, on note $\delta(q, a) = q'$ si $(q, a, q') \in \delta$. De plus, pour un mot $u \in \Sigma^\star$, $\delta(q, u) = \delta(\delta(q, a), v)$ où $u = av$ et $a \in \Sigma$.

Théorème 23 (Théorème de déterminisation).

Les langages reconnus par les automates finis déterministes sont exactement ceux reconnus par les automates finis non déterministes.

Démonstration. Il est évident que la classe des langages reconnus par les automates finis déterministes est inclus dans la classe des langages reconnus par les automates finis non déterministes.

Réciproquement, on applique un algorithme de déterminisation.

Étant donné un automate fini non déterministe N, on construit un automate fini déterministe A qui reconnait le même langage.

On fixe $\Sigma(A) = \Sigma(N)$, $Q(A) = \mathcal{P}(Q(N))$ et $q_0(A) = \{q_0(N)\}$.

De plus, $\forall E \in Q(N), E \in F(N) \Leftrightarrow (\exists q \in E : q \in F(A))$.

Il reste à définir $\Sigma(N)$.

La fonction de transition de A est définie, pour $q \in Q(A)$ et $a \in \Sigma$, par

$$(q, a) \to \left\{q' \in Q(N) \,\middle|\, \exists s \in q : (s, a, q') \in \delta(N)\right\}$$

Cet automate est déterministe par construction et reconnait le même langage que N puisque les configuration successives du calcul de A décrivent tous les calculs possibles de N sur la même entrée. □

23.4 Les expressions régulières

La faible puissance des automates finis permet d'exprimer les langages qu'ils reconnaissent avec des expressions simples. On introduit ici ce genre de formules.

Définition 81 (Expression régulière)**.**

Soit Σ, un alphabet.
Une expression régulière sur Σ est une expression qui satisfait la définition inductive suivante

- $\emptyset$ est une expression régulière.
- ε est une expression régulière.
- $a \in \Sigma$ est une expression régulière.
- Si M et N sont des expressions régulières, alors $M + N$, MN et $M^\star$ aussi.

De façon intuitive, une expression régulière représente un langage. Le langage est alors dit régulier.

Définition 82 (Langage régulier)**.**

Le langage $\mathcal{L}(E)$ engendré par une expression régulière E est défini par induction :

- $\mathcal{L}(\emptyset) = \emptyset$
- $\mathcal{L}(\varepsilon) = \{\varepsilon\}$
- $\mathcal{L}(a) = \{a\}$
- $\mathcal{L}(MN) = \mathcal{L}(M) \cdot \mathcal{L}(N)$
- $\mathcal{L}(M + N) = \mathcal{L}(M) + L(N)$
- $\mathcal{L}(M^\star) = \mathcal{L}(M)^\star$

Notation 33.

On note $\mathfrak{R}(\Sigma)$, les expressions régulières sur Σ.

23.5 Les automates finis non déterministes généralisés

Définition 83 (Automates finis non déterministes généralisés)**.**

Un automate fini généralisé est un 5-uplet $(Q, \Sigma, q_0, \delta, q_f)$ où
- Q est un ensemble fini d'états,
- Σ est l'alphabet de l'automate,
- $q_0 \in Q$ est l'état initial,
- $q_f \in Q$ est l'état acceptant,
- $\delta : (Q \setminus \{q_f\}) \times (Q \setminus \{q_0\}) \to \mathfrak{R}(\Sigma)$.

La notion d'automate fini déterministe généralisée n'est pas pertinente pour des raisons qui vont apparaitre.

Soit u et A un automate non déterministe généralisé.

A accepte u s'il existe des mots $u_1, \dots, u_k$ et des états $q_1, \dots, q_k$ tels que $\forall i \in [\![1, k]\!], u_i \in \mathcal{L}(\delta(q_{i-1}, q_i))$ et $q_k = q_f$.

Une transition ne lit pas seulement une lettre mais tout un sous mot décrit par une des expressions régulières qui étiquette les transitions possibles.

Proposition 85.

Pour tout automate fini non déterministe, il existe un automate fini généralisé équivalent.

Démonstration. Soit A un automate fini non déterministe. On construit G, un automate fini non déterministe généralisé, équivalent à A.

Pour obtenir G à partir de A, on ajoute un état initial q_0, un état final q_f et des ε-transitions de q_0 vers l'état initial de A et des états finaux de A vers q_f.

Entre deux états q_i et q_j, s'il manque une transition, on ajoute une transition vide ($\emptyset$).

Enfin, on fusionne les transitions redondantes. □

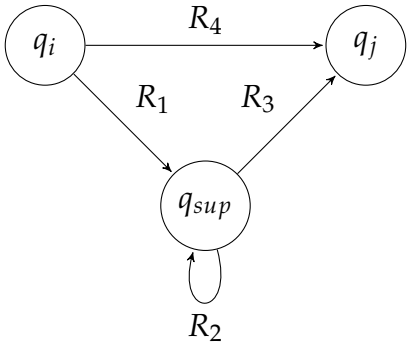
Proposition 86.

Étant donné un automate fini généralisé avec au moins trois états, il existe un automate fini non déterministe généralisé équivalent avec un état de moins.

Démonstration. On choisit un état $q_{sup} \notin \{q_0, q_f\}$.

On se donne une paire d'états (q_i, q_j), $q_i \neq q_{sup}$ et $q_j \neq q_{sup}$.

On note $R_1 = \delta(q_i, q_{sup})$, $R_2 = \delta(q_{sup}, q_{sup})$, $R_3 = \delta(q_{sup}, q_j)$ et $R_4 = \delta(q_i, q_j)$. On fixe $R = R_4 + (R_1 R_2^\star R_3)$ et on fixe $\delta(q_i, q_j) = R$.



On répète le processus pour toute paire d'états. Ensuite, on supprime q_{sup}. L'automate obtenu a un état de moins et reconnait le même langage. □

Définition 84.

On appelle langage rationnel, un langage reconnu par un automate fini.

Théorème 24 (Théorème de KLEENE)**.**

Les langages rationnels sont exactement les langages réguliers.

Démonstration. Soit A un automate fini. On crée un automate généralisé G équivalent (cf. prop. 85). Par récurrence, on construit un automate généralisé à deux états (q_0 et q_f) équivalent (cf. prop. 86). La transition de q_0 à q_f, est étiqueté par une expression régulière qui est le langage de l'automate A.

Réciproquement, on se donne une expression régulière. On construit par induction un automate fini asynchrone qui reconnait le langage de l'expression régulière. Pour cela, on utilise les construction suivantes.

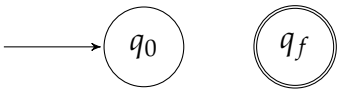


FIGURE 23.1 – Automate reconnaissant ∅

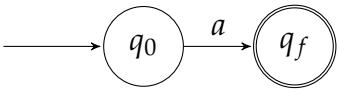


FIGURE 23.2 – Automate reconnaissant $\{a\}$

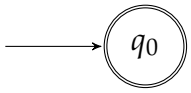


FIGURE 23.3 – Automate reconnaissant ε

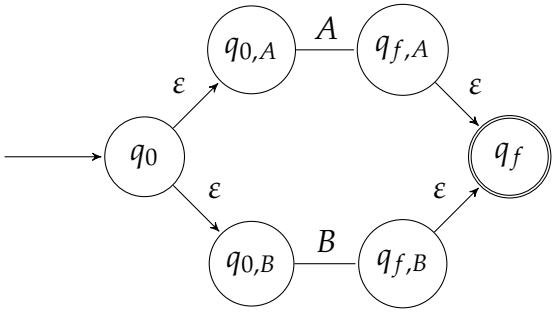


FIGURE 23.4 – Automate reconnaissant $A + B$

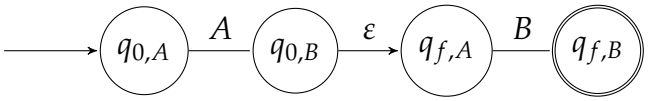


FIGURE 23.5 – Automate reconnaissant AB

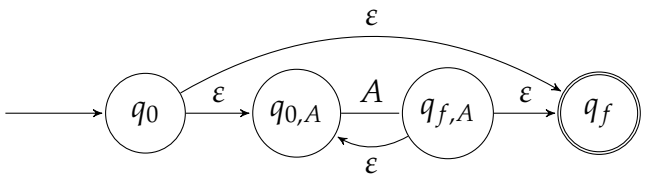


FIGURE 23.6 – Automate reconnaissant $A^\star$

Ainsi, on construit inductivement un automate pour toute expression régulière. □

On parle ainsi indistinctement de langage régulier ou rationnel pour désigner cette classe de langage.

À cause de la simplicité des expressions régulières, on voit bien que pour obtenir des mots arbitrairement long, on ne peut compter que sur une étoile. Or, les expressions régulières étant très locales, on peut itérer une étoile indépendamment du reste. Donc dans tout mot suffisamment long, on peut trouver un motif, correspondant à l'expression sous l'étoile, qui peut être itéré pour obtenir une famille de mots arbitrairement longs qui sont tous dans ce même langage rationnel.

La contraposée est aussi intéressante : si en répétant un motif, on sort du langage, celui ci ne pouvait pas être rationnel.

Le théorème suivant formalise cette intuition.

Théorème 25 (Lemme de l'étoile)**.**

Soit L un langage rationnel. Il existe un entier p tel que tout mot s de L de longueur $|s| \geqslant p$ possède une factorisation $s = xyz$ telle que
- $0 < |y|$
- $|xy| \leqslant p$
- $\forall i \in \mathbb{N}, xy^i z \in L$

Démonstration. Soit $A = (Q, \Sigma, q_0, \delta, F)$ un automate fini déterministe qui reconnait L.

On pose $p = |Q|$.

Soit $s \in L$ tel que $|s| \geqslant p$. On pose $n = |s|$. Soit $r_0, \ldots, r_n$ la suite d'états du calcul de A sur s. On a $r_0 = q_0$, $r_n \in F$ et $\forall i \in [\![1, n]\!], \delta(r_{i-1}, s_{i-1}) = r_i$.

Parmi les $p+1$ premiers états de cette suite, au moins deux sont identiques. On a $r_j = r_l$ avec $j < l \leqslant p+1$. On pose

$$x = s_0 \ldots s_{j-1}$$
$$x = s_j \ldots s_{l-1}$$
$$x = s_l \ldots s_{n-1}$$

$|xy| = l - 1 \leqslant p$. On a aussi $\forall i \in \mathbb{N}, xy^i z \in L$ puisqu'il y a un calcul acceptant de l'automate. □

On utilise ce théorème, le plus souvent, pour nier le fait qu'un langage est rationnel.

23.6 Minimisation

Il est fréquemment utile d'utiliser des expressions régulières pour chercher un motif dans un texte. La recherche elle même est alors faite grâce à un automate fini. Cependant, la méthode de construction des automate ne donne pour l'instant pas d'automate simple. En effet, ils sont soit non déterministes (donc avec de multiples états courants), soit déterminisés (mais avec un ensemble d'états très grands). Pour des raisons de performance, il est judicieux de chercher le plus petit automate déterministe dont le langage reconnu est celui exprimé pas l'expression régulière. On parle de minimisation d'automate.

Un automate est dit minimal si son nombre d'états est minimal. Ainsi, on n'a pas a priori d'unicité.

L'idée de la minimisation est de fusionner tous les états qui mène tous deux au même résultat (acceptation ou refus) pour toute suite de lettres. Ces deux états sont en effet indistinguables.

Définition 85.

Soit L un langage sur Σ. Soit x et y, deux mots sur Σ. On dit que x et y sont indistinguables par rapport au langage L si $\forall z \in \Sigma^\star, xz \in L \Leftrightarrow yz \in L$.

Notation 34.

Si x et y sont indistinguables par rapport au langage L, on note

$$x \equiv_L y$$

Proposition 87.

$\equiv_L$ est une relation d'équivalence.

Démonstration. On vérifie facilement les hypothèses d'une relation d'ordre. □

Définition 86.

Le nombre de classes d'équivalence de $\equiv_L$ est appelé l'indice de L.

Théorème 26 (Théorème de MYHILL-NERODE)**.**

L est rationnel si et seulement si $\equiv_L$ a un nombre fini de classes d'équivalence.
Dans ce cas, le nombre de classes d'équivalence est égal au nombre d'états du plus petit automate fini déterministe reconnaissant L.

Démonstration. ($\Rightarrow$) : Soit $A = (Q, \Sigma, q_0, \delta, F)$ un automate fini déterministe reconnaissant L. On note $k = |Q|$.

Soit $(x, y) \in (\Sigma^\star)^2$. Si $\delta q_0, x) = \delta(q_0, y)$ alors $x \equiv_L y$.

En effet

$$\begin{aligned} \forall z \in \Sigma^\star, xz \in L &\Leftrightarrow \delta(q_0, xz) \in F \\ &\Leftrightarrow \delta(\delta(q_0, x), z) \in F \\ &\Leftrightarrow \delta(\delta(q_0, y), z) \in F \\ &\Leftrightarrow \delta(q_0, yz) \in F \\ &\Leftrightarrow yz \in L \end{aligned}$$

($\Leftarrow$) : On suppose L d'indice k et on construit A à k états à partir de $\equiv_L$.
Les états de A sont $L/\equiv_L$ et la fonction de transition $\delta(\overline{x}, a) = \overline{xa}$.
δ est bien définie : si $\overline{x} = \overline{y}$ alors $\overline{xa} = \overline{ya}$. En effet,

$$\begin{aligned} \forall z \in \Sigma^\star, (xa)z \in L &\Leftrightarrow x(az) \in L \\ &\Leftrightarrow y(az) \in L \\ &\Leftrightarrow (ya)z \in L \end{aligned}$$

On prend $\overline{\varepsilon}$ comme état initial et $\{\overline{x} \mid x \in L\}$ comme ensemble des états finaux. □

Remarque 12.

Toute classe d'équivalence est composée soit de mots de L, soit de mots de $\Sigma^\star \setminus L$.
A reconnait L : Soit $x \in \Sigma^\star, \delta(\overline{\varepsilon}, x) = \overline{\varepsilon x} = \overline{x}$. On a donc $\overline{x} \in F \Leftrightarrow x \in L$.

Définition 87.

Soit $A = (Q, \Sigma, q_0, \delta, F)$ un automate fini déterministe. Un état q est dit accessible s'il existe un mot u tel que $\delta(q_0, u) = q$. Un état est dit co-accessible s'il existe un mot u tel que $\delta(q, u) \in F$.
Un automate est dit émondé si tous ses états sont accessibles et co-accessibles.
Un automate est dit complet s'il existe une transition pour tout élément de $Q \times \Sigma$.

Définition 88.

Soit $A = (Q, \Sigma, q_0, \delta, F)$ un automate fini déterministe reconnaissant L.
On suppose que tous les états sont accessibles depuis q_0.
On définit la relation d'équivalence sur Q

$$q = q' :\Leftrightarrow \forall x \in \Sigma^\star, \delta(q, x) \in F \Leftrightarrow \delta(q', x) \in F$$

On définit un automate quotient $\overline{M} = (\overline{Q}, \Sigma, \overline{q_0}, \overline{\delta}, \overline{F})$ où
— $\overline{Q} = (Q/\equiv)$
— $\overline{F} = \{\overline{q} \mid q \in F\}$
— $\overline{\delta} : (\overline{q}, a) \mapsto \overline{\delta(q, a)}$

Théorème 27.

$\overline{M}$ est un automate minimal reconnaissant L.

Démonstration. Soit $x \in \Sigma^\star$, $\overline{\delta}(\overline{q_0}, x) = \overline{\delta(q_0, x)} \Leftrightarrow \delta(q_0, x) \in F \Leftrightarrow x \in L$.

Minimalité de $\overline{M}$: On applique le théorème de MYHILL-NERODE. Il suffit d'exhiber $\left|\overline{Q}\right|$ mots non équivalents par $\equiv_L$.

Pour chaque $\overline{q} \in \overline{Q}$ on a un mot $u_q \in \Sigma^\star$ tel que $\delta(q_0, u_q) = q$. Ces $\left|\overline{Q}\right|$ mots sont non équivalent par $\equiv$.

Contraposée : si $u_q \equiv u_r$ alors $q \equiv r$.

En effet, soit $x \in \Sigma^\star$, $\delta(q, x) \in F \Leftrightarrow \delta(q_0, u_q x) \in F \Leftrightarrow u_q x \in L \Leftrightarrow u_r x \in L$ car $u_q \equiv_L u_r$.

$$\delta(r, x) \in F \Leftrightarrow u_r x \in L$$

On construit un graphe orienté dont les sommets sont des couples de Q^2 et tel qu'il existe un arc étiqueté par a entre (q, q') et (r, r') si $r = \delta(q, a)$ et $r' = \delta(q', a)$.

Pour distinguer q de q', on doit trouver un chemin de (q, q') vers $(r, r') \in F \times (Q \setminus F) \cup (Q \setminus F) \times F$. Un tel chemin peut être de longueur au plus $|Q|^2 - 1$. □

Chapitre 24

Les automates à pile

Nous avons déjà introduit et travaillé avec des automates à plusieurs piles mais les automates avec une pile unique sont plus répandus. Il existe deux types d'automates à pile : les déterministes, qui sont la simple restriction à une pile de ce qu'on a déjà vu, et les non déterministes, compris entre ces deux modèles.

24.1 Les automates à pile non déterministes

Définition 89 (Automate à pile non déterministe)**.**

Un automate à pile non déterministe est un 7-uplet $(Q, \Sigma, \mathcal{Z}, q_0, z_0, \delta, F)$, où :

- Q est l'ensemble d'états
- Σ est l'alphabet d'entrée
- $\mathcal{Z}$ est l'alphabet de pile
- q_0 est l'état initial
- z_0 est le symbole de fond de pile
- $\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times \mathcal{Z} \times Q \times \mathcal{Z}^\star$ est la fonction de transition
- F est l'ensemble des états acceptants

Un automate à pile est un modèle d'automate dont la mémoire est organisée sous forme de pile : l'automate ne peut lire et écrire qu'en haut de la pile. Ce modèle a donc une mémoire illimitée mais avec des contraintes sur les accès.

Une configuration de l'automate est un triplet $(q, w, h) \in Q \times \Sigma^\star \times \mathcal{Z}^\star$. La première composante est l'état interne de l'automate, la seconde est le mot restant en entrée et la dernière est l'état de la pile.

Un calcul de l'automate sur un mot $u \in \Sigma^\star$ est une suite de transitions à partir de la configuration initiale (q_0, u, z_0). Il y a transition de la configuration (q, yw, zh), où $y \in \Sigma \cup \{\varepsilon\}$ et $z \in \mathcal{Z}$ vers la configuration $(q', w, h'h)$ lorsque $(q', h') \in \delta(q, y, z)$ et on écrit :

$$(q, yw, zh) \to (q', w, h'h)$$

Lorsque $y = \varepsilon$, le mot d'entrée ne change pas. On parle alors d'une ε-transition ou d'une transition spontanée ou asynchrone. Avec ce modèle, pour qu'une transition soit possible, la pile ne doit pas être vide.

On dit qu'un mot est accepté par l'automate s'il existe une série de transitions qui conduit à une configuration acceptante. Il y a plusieurs façon de déterminer une configuration acceptante :

— Reconnaissance par pile vide : les configurations acceptantes sont les configurations de la forme $(q, \varepsilon, \varepsilon)$ où $q \in Q$. Auquel cas, la composante F est inutile.
— Reconnaissance par état final : les configurations acceptantes sont les configurations de la forme (q, ε, h) où $q \in F$ est un état final.
— Reconnaissance par pile vide et état final : les configurations acceptantes sont les configurations de la forme $(q, \varepsilon, \varepsilon)$ où $q \in F$ est un état final.

Le langage de l'automate est l'ensemble des mots de $\Sigma^\star$ qui sont acceptés.
Les trois modes d'acceptation sont équivalents.

Notation 35.

L'ensemble des langages reconnus par automate à pile non déterministe est noté **AP**.

24.2 Les automates à pile déterministes

Un automate à pile déterministe est simplement la restriction à 1 pile des automates à plusieurs piles.

De façon surprenante et à la différences des automates finis, les automates à pile déterministes n'ont pas la même puissance que les automates à pile non déterministes.

Définition 90 (Automate à pile déterministe)**.**

Un automate à pile déterministe est un 7-uplet $(Q, \Sigma, \mathcal{Z}, q_0, z_0, \delta, F)$, où :
- Q est l'ensemble d'états
- Σ est l'alphabet d'entrée
- $\mathcal{Z}$ est l'alphabet de pile
- q_0 est l'état initial
- z_0 est le symbole de fond de pile
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \mathcal{Z} \to Q \times \mathcal{Z}^\star$ est la fonction de transition
- F est l'ensemble des états acceptants

Le fonctionnement est le même que pour les automates à piles non déterministes mais une seule transition est possible à chaque étape, il existe donc une unique suite de configuration pour une entrée donnée.

24.3 Variantes

24.3.1 Modes d'acceptation

On peut définir des configurations interne d'acceptations moins régulières que les trois caractérisations précédemment données.

24.3.2 Automate synchrone

Définition 91.

Un automate à pile est dit synchrone s'il ne possède pas d'ε-transition.

Définition 92.

Un automate à pile des simple s'il ne possède qu'un seul état.

Théorème 28.

Tout langage de **AP** peut être reconnu par un automate synchrone simple.

24.3.3 Langage de pile

Définition 93.

Le langage de pile d'un automate à pile est l'ensemble des mots qui apparaissent sur la pile lors d'un calcul réussi.

Théorème 29.

Tout langage de pile est rationnel.

24.4 Propriétés

Théorème 30.

L'équivalence de deux automate à pile déterministe est décidable.

Chapitre 25

Les automates linéairement bornés

25.1 Définition

Définition 94 (Machine de TURING [Car08a]).

Un automate linéairement borné est un 8-uplet $(Q, \Gamma, B, \Sigma, q_0, \delta, F, k)$ où :
- Q est un ensemble fini d'états,
- Γ est l'alphabet de travail,
- $B \in \Gamma$ est un symbole particulier dit symbole blanc,
- Σ est l'alphabet des symboles en entrée ($\Sigma \subseteq \Gamma \setminus \{B\}$)
- $q_0 \in Q$ est l'état initial,
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\longleftarrow, \longrightarrow, \texttt{HALT}\}$ est la fonction de transition,
- $F \subseteq Q$ est l'ensemble des états acceptants,
- $k \in \mathbb{N}^*$.

Ce modèle de calcul est similaire aux machines de TURING à une différence près : le ruban est de taille fini et, plus précisément, proportionnel à la taille du mot en entrée avec un coefficient k.

En pratique, on bloque le déplacement en dehors du ruban. On peut aussi supposer qu'il y a une case spéciale à chaque extrémité comportant des symboles spéciaux signalant les extrémités du ruban à l'automate.

25.2 Propriétés

Théorème 31.

L'arrêt d'un automate linéairement borné est décidable.

Démonstration. Il n'existe qu'un nombre fini de configuration sur une entrée donnée. Une fois ce nombre de configurations atteint, soit la machine s'est arrêté, soit elle parcourt un cycle, dans ce cas, la machine ne termine pas. □

Chapitre 26

La hiérarchie de CHOMSKY

Une approche pour définir un langage consiste à en donner des règles de construction. Une façon de faire cela, dans un cas très général, sont les grammaires. Une grammaire est un ensemble de règles qui, en partant d'un unique symbole permettent de générer des mots par des règles de substitutions.

Définition 95 (Grammaire)**.**

Une grammaire est un 4-uplet $(\mathcal{T}, \mathcal{N}, \mathcal{R}, \mathcal{S})$ où
- $\mathcal{T}$ un ensemble fini de symboles terminaux, appelé alphabet terminal,
- $\mathcal{N}$ un ensemble fini de symboles non terminaux (parfois appelé aussi alphabet des variables),
- $\mathcal{R} \in \mathcal{P}\left(\left((\mathcal{T} \cup \mathcal{N})^\star\right)^2\right) \setminus \{\varepsilon\}$ un ensemble fini de règles,
- $\mathcal{S} \in \mathcal{N}$ l'axiome.

On note $\mathcal{V} = \mathcal{T} \cup \mathcal{N}$.

Une règle (x, y) est aussi notée $x \to y$.

On remarque qu'il y a deux types de symboles : les terminaux et les non-terminaux. Les mots générés en appliquant des règles à partir de l'axiome vont êtres constitués de ces deux types de lettres, mais ce qu'on appelle le langage engendré par la grammaire va être des mots uniquement formés de symboles terminaux. Les non-terminaux doivent encore être remplacés en utilisant différentes règles. Cette procédure explique le nom de ces deux classes : si un mot contient au moins un symbole non-terminal, ce mot ne fait pas partie du langage, car la dérivation n'est pas terminée.

Définition 96.

u dérive immédiatement en y par l'emploi de la règle $x \to y$ si

$$\exists (p, s) \in (V^\star)^2 : u = pxs \wedge v = pys$$

En d'autres termes, une dérivation immédiate du mot u en le mot v consiste à remplacer, dans u, une occurrence de x par y.

Notation 36.

Si v dérive immédiatement de u, on écrit $u \to v$. On désigne comme d'habitude les différentes fermetures.

Définition 97.

On appelle dérivation la fermeture réflexive et transitive de la dérivation immédiate.

Définition 98.

Soit $\mathcal{G}$ une grammaire.
Le langage engendré par $\mathcal{G}$ est

$$\{u \in \mathcal{T}(\mathcal{G})^\star \mid \mathcal{S}(\mathcal{G}) \to^* u\}$$

Notation 37.

On note $L(\mathcal{G})$ le langage engendré par $\mathcal{G}$.

La hiérarchie de CHOMSKY est une classification des grammaires formelles et des langages ainsi engendrés. On définit 4 classes de langages notées L_0, L_1, L_2 et L_3 qui correspondent à différentes restrictions sur les règles. Plus il y a de restrictions, plus la grammaire sera simple. Selon ce niveau, les langages générés sont plus ou moins riches et sont reconnus par des machines de complexité distinctes.

26.1 Les grammaires générales

Définition 99.

Une grammaire générale est une grammaire dont les règles sont de la forme

$$a \to \beta$$

où $\alpha \in \mathcal{V}^\star \mathcal{N} \mathcal{V}^\star$ et $\beta \in \mathcal{V}^\star$
Aucune restriction n'est imposé aux règles.

Notation 38.

L'ensemble des langages engendrés par les grammaires générales est noté L_0.

Théorème 32.

$$L_0 = \mathbf{RE}$$

Démonstration.

$L_0 \subseteq \mathbf{RE}$: Soit $\mathcal{G}$ une grammaire générale et $u \in T(\mathcal{G})^\star$.

On crée une machine de TURING qui reconnait le mot u si et seulement s'il appartient à $L(\mathcal{G})$. Pour cela, on prend une machine de TURING non déterministe qui exécute à l'envers toutes la règles possibles. La machine termine si elle atteint l'axiome.

Si la machine termine, alors on a une suite de règle qui permet de dériver u depuis l'axiome. Réciproquement, si une dérivation existe, elle est nécessairement explorée lors de l'exécution de la machine de TURING non déterministe.

$\mathbf{RE} \subseteq L_0$: soit M une machine de TURING. On note par le mot

$$\mu a_1 \dots \ a_{i-1} \ (a_i, q) \ a_{i+1} \ \dots a_n \mu$$

une configuration de M où le ruban est $a_1 \dots a_n$, l'état courant est q et la tête de lecture se trouve sur a_i. Le symbole μ symbolise la fin du ruban et ne fait pas parti de l'alphabet de M.

On considère les symboles du ruban comme terminaux ainsi que les symboles de

$$\left\{ \overline{(a,q)} \;\middle|\; (a,q) \in \Gamma(M) \times Q(M) \right\}$$

et les symboles de

$$\{(a,q) \mid (a,q) \in \Gamma(M) \times Q(M)\}$$

comme non terminaux.

En exécutant la transition $(q, a_i) \to (q', a_i', \longrightarrow)$, on passe de la configuration

$$\mu a_1 \dots \; a_{i-1} \; (a_i, q) \; a_{i+1} \; \dots a_n \mu$$

à la configuration

$$\mu a_1 \dots \; a_i' \; (a_{i+1}, q') \; a_{i+2} \dots a_n \mu$$

On peut exécuter cette transition en appliquant la règle

$$(a_i, q) \; a_{i+1} \to a_i' \; (a_{i+1}, q')$$

à ces configuration.

S'il n'y a pas de symbole à droite (ruban blanc inexploré), on prend la règle $(a_i, q) \; \mu \to a_i' \; (B(M), q')\mu$.

De même une transition de la forme $(q, a_i) \to (q', a_i', \longleftarrow)$ se traduit en la règle $a_{i-1} \; (a_i, q) \to (a_{i-1}, q')a_i'$ et $\mu \; (a_i, q) \to \mu(B(M), q')a_i'$.

Pour les transitions de la forme $(q, a_i) \to (q', a_i', \texttt{HALT})$, on ajoute les règles $(a, q) \to \overline{(a, q)}$.

Ainsi, on simule l'exécution de la machine de TURING grâce à une grammaire générale. On peut ainsi construire l'ensemble des langages récursivement énumérables par les grammaires générales. □

Corollaire 22.

Le problème de l'appartenance d'un mot à un langage de cette classe est indécidable.

Démonstration. En effet, l'arrêt d'une machine de TURING est indécidable. □

Remarque 13.

Le problème de l'appartenance est certes indécidable, mais il est reconnaissable. En effet, il existe une machine de TURING qui va terminer exactement sur tous les mots du langage.

26.2 Les grammaires contextuelles

Définition 100.

Une grammaire contextuelle est une grammaire dont les règles sont de la forme

$$\alpha A \beta \to \alpha \gamma \beta$$

où $A \in \mathcal{N}$, $(\alpha, \beta, \gamma) \in (\mathcal{V}^\star)^3$ et $\gamma \neq \varepsilon$.
Les règles sont sensibles au contexte des symboles non terminaux mais ne peuvent pas le changer.

Notation 39.

L'ensemble des langages engendrés par les grammaires contextuelle est noté L_1.

Théorème 33.

$$L_1 = \mathbf{ALB}$$

Démonstration. $L_1 \subseteq \mathbf{ALB}$: Soit $\mathcal{G}$ une grammaire contextuelle et $u \in T(\mathcal{G})^\star$.

On crée un automate linéairement borné qui reconnait le mot u si et seulement s'il appartient à $L(\mathcal{G})$. Pour cela, on prend un automate linéairement borné non-déterministe qui exécute, à l'envers, toutes la règles possibles. On respecte la restriction de la mémoire puisque les règles ne peuvent pas allonger le mot. Le mot est donc de longueur décroissante au cours du calcul. La machine termine si elle atteint l'axiome.

Si la machine termine, alors on a une suite de règle qui permet de dériver u depuis l'axiome. Réciproquement, si une dérivation existe, elle est nécessairement explorée lors de l'exécution de l'automate linéairement borné non déterministe.

$\mathbf{ALB} \subseteq L_1$: Soit M un automate linéairement borné. On note par le mot $\mu a_1 \dots a_{i-1} (a_i, q) a_{i+1} \dots a_n \mu$ une configuration de M où le ruban est $a_1 \dots a_n$, l'état courant est q et la tête de lecture se trouve sur a_i. Le symbole μ symbolise les extrémités du ruban et ne fait pas parti de l'alphabet de M.

On considère tous les symboles introduits comme non terminaux, à l'exception de μ (qui doit rester inchangé puisque le ruban est de taille constante.

Le but est de définir un ensemble de règles qui permettent de simuler la machine. On va construire un ensemble de règles qui se regroupent en séquences simulant à chaque fois une transition. Une séquence commencée ne peut pas être interrompu. Entre deux séquences, plusieurs règles sont possibles et elles correspondent aux transition possible de l'automate.

On fait une disjonction de cas.

En exécutant la transition $(q, a_i) \to (q', a'_i, \longrightarrow)$, on passe de la configuration

$$\mu a_1 \dots \ a_{i-1} \ (a_i, q) \ a_{i+1} \ \dots a_n \mu$$

à la configuration

$$\mu a_1 \dots \ a'_i \ (a_{i+1}, q') \ a_{i+2} \dots a_n \mu$$

Pour simuler cette transition, on applique la règle non-contextuelle $(a_i, q) \to ((a'_i, q'), \longleftarrow)$ pour atteindre

$$\mu a_1 \dots \ a_{i-1} \ ((a'_i, q'), \longrightarrow) \ a_{i+1} \ \dots a_n \mu$$

Le nouvel ensemble de symbole introduit ne contient que des symboles non terminaux.

On utilise ensuite la règle $((a'_i, q'), \longrightarrow)a_{i+1} \to ((a'_i, q'), \longrightarrow)(a_{i+1}, q')_N$ pour obtenir

$$\mu a_1 \dots \ a_{i-1} \ ((a'_i, q'), \longrightarrow) \ (a_{i+1}, q')_N \ \dots a_n \mu$$

Ensuite, on doit supprimer l'information de la cellule précédente avec la règle $((a'_i, q'), \longrightarrow) \ (a_{i+1}, q')_N \to a'_i \ (a_{i+1}, q')_N$

$$\mu a_1 \dots \ a_{i-1} \ a'_i \ (a_{i+1}, q')_N \ \dots a_n \mu$$

Ensuite, on supprime la marque sur la nouvelle case afin de finir l'exécution de la transition grâce à la règle $a'_i \ (a_{i+1}, q')_N \to a'_i \ (a_{i+1}, q')$

$$\mu a_1 \dots \ a'_i \ (a_{i+1}, q') \ a_{i+2} \dots a_n \mu$$

La transition est alors réalisée et on peut passer à la suivante.

De même une transition de la forme $(q, a_i) \to (q', a'_i, \longleftarrow)$ se traduit en les règles

$$\begin{aligned}
(a_i, q) &\to ((a'_i, q'), \longleftarrow) \\
a_{i-1} \ ((a'_i, q'), \longleftarrow) &\to (a_{i-1}, q')_N((a'_i, q'), \longleftarrow) \\
(a_{i+1}, q')_N \ ((a'_i, q'), \longleftarrow) \ &\to (a_{i+1}, q')_N \ a'_i \\
a'_i \ (a_{i+1}, q')_N &\to a'_i \ (a_{i+1}, q')
\end{aligned}$$

Pour les transitions de la forme $(q, a_i) \to (q', a_i', \texttt{HALT})$, on ajoute les règles $(a, q) \to \overline{(a, q)}$ où le nouveau type d'état est terminal.

Ensuite, on propage la terminaison avec les règles

$$\begin{aligned} a_{i-1}\ \overline{(q,a)} &\to \overline{a_{i-1}}\ \overline{(q,a)} \\ \overline{(q,a)}\ a_{i+1} &\to \overline{(q,a)}\ \overline{a_{i+1}} \\ a_{i-1}\ \overline{a_i} &\to \overline{a_{i-1}}\ \overline{a_i} \\ \overline{a_i}\ a_{i+1} &\to \overline{a_i}\ \overline{a_{i+1}} \end{aligned}$$

À la fin du calcul, le ruban est de la forme

$$\mu\overline{a_1} \dots \overline{a_{i-1}}\ \overline{(a_i, q)}\ \overline{a_{i+1}}\ \dots \overline{a_n}\mu$$

qui est un mot de symboles terminaux. Ainsi, on simule l'exécution d'un automate linéairement bornée grâce à une grammaire contextuelle. □

Corollaire 23.

Le problème de l'appartenance d'un mot à un langage de cette classe est décidable.

Démonstration. En effet, l'arrêt d'un automate linéairement borné est décidable. □

26.3 Les grammaire algébriques

Définition 101.

Une grammaire non-contextuelle (dite aussi algébrique ou acontextuelle) est une grammaire dont les règles sont de la forme

$$A \to \gamma$$

où $A \in \mathcal{N}$, $\gamma \in \mathcal{V}^\star$.
Les règles sont insensibles au contexte des symboles non terminaux : elles agissent de la même façon quel que soit le contexte.

Notation 40.

L'ensemble des langages engendrés par les grammaires algébriques est noté L_2.

Théorème 34.

$$L_2 = \mathbf{AP}$$

Démonstration. $L_2 \subseteq \mathbf{AP}$: soit $\mathcal{G}$ une grammaire algébrique et $u \in T(\mathcal{G})^\star$. On construit un automate à pile A qui cherche à déterminer de façon non déterministe une dérivation de u.

Soit $w \in T(\mathcal{G})^\star$ la partie de u non encore lu. On suppose que A utilise \$ comme symbole de fond de pile. On note $s = t\$ \in V(\mathcal{G})^\star \cdot \{\$\}$ le mot de la pile.

A doit accepter si et seulement si $t \to^* w$.

Initialement, on part de $t = S(\mathcal{G})$. On distingue deux cas.

1$^{\text{er}}$ cas : le sommet de pile est un symbole non terminal

$$s = Bt\$ \in N(\mathcal{G}) \cdot V(\mathcal{G})^\star \cdot \{\$\}$$

L'automate dépile B. On choisit une règle $B \to v$, puis empile v.

2$^{\text{ème}}$ cas : le symbole de pile est un symbole terminal.

$$s = at\$ \in T(\mathcal{G}) \cdot V(\mathcal{G})^\star \cdot \{\$\}$$

L'automate dépile a. Si a est la première lettre de w (la prochaine lettre lue en entrée). Si on lit un autre lettre, le calcul échoue.

AP $\subseteq L_2$: Soit A un automate à pile.
On peut supposer sans perte de généralité que
— A a un unique état acceptant q_f,
— A vide toujours sa pile avant d'accepter,
— chaque transition de P empile ou dépile un symbole, mais pas les deux.
On construit une grammaire $\mathcal{G}$. Pour toute paire d'état $(p, q) \in Q(A)^2$, on crée une variable $A_{p,q}$ qui engendre exactement les mots $w \in \Sigma(\mathcal{G})^\star$ tel que $\delta(A)(p, w, \varepsilon)$ contient (q, ε).
On remarque deux choses :
— Si $\delta(A)(p, w, \varepsilon)$ contient (q, ε) alors $\forall \gamma \in \Gamma(\mathcal{G})^\star, \delta(A)(p, w, \gamma)$ contient (q, γ).
— Réciproquement, si $\delta(A)(p, w, \gamma)$ contient (q, γ) et que γ n'est jamais dépilé au cours du calcul $\delta(A)(p, w, \varepsilon)$ contient (q, ε)
On prend comme axiome de $\mathcal{G}$, l'état A_{q_0,q_f}.
Supposons que $\delta(A)(p, w, \varepsilon)$ contient (q, ε). A commence par empile un symbole t. On distingue deux cas :
— Ce symbole n'est dépilé qu'à la dernière transition. La règle correspondante est $A_{p,q} \to aA_{r,s}b$ avec
  — a, première lettre de w,
  — b, dernière lettre de w,
  — r, état qui suit p au cours du calcul,
  — s, état qui précède q au cours du calcul.
— Ce symbole est dépilé au cours du calcul, la pile devient vide en atteignant un certain état r. On introduit la règle $A_{p,q} \to A_{p,r}A_{rq}$.
Ceci mène aux règles
— $A_{p,q} \to aA_{r,s}b$, si $\delta(A)(q, a, \varepsilon)$ contient (r, t) et $\delta(A)(s, b, t)$ contient (q, ε) pour tout $t \in \Gamma(A)$.
— $A_{p,q} \to A_{p,r}A_{r,q}$ pour tout $(p, q, r) \in Q(A)^3$.
— $A_{p,p} \to \varepsilon$.

Proposition 88.

Si $A_{p,q} \to^* x$ avec $x \in \Sigma(A)^\star$, alors $\delta(p, x, \varepsilon)$ contient (q, ε).

Démonstration. Par récurrence sur la longueur de la dérivation k de x.

$k = 1$: la dérivation est de la forme $A_{p,p} \to \varepsilon$. On a bien $\delta(p, \varepsilon, \varepsilon)$ contient (p, ε).

Supposons le résultat vrai pour toute dérivation de longueur inférieure à k. Considérons le cas $k + 1$.

On regarde la première règle appliquée au cours de cette dérivation.

— La règle est de la forme $A_{p,q} \to aA_{r,s}b$. On a $A_{r,s} \to^* y$ (une dérivation de longueur inférieur à k) avec $x = ayb$, par hypothèse de récurrence $\delta(A)(r, y, \varepsilon)$ contient (s, ε).
Par construction de cette règle, $\delta(A)(p, a, \varepsilon)$ contient (r, t) et $\delta(A)(s, b, t)$ contient (q, ε).

— $A_{p,q} \to A_{p,r}A_{r,q}$ avec $A_{p,r} \to^* y$, $A_{r,q} \to^* z$ et $x = yz$. Par hypothèse de récurrence $\delta(A)(p, y, \varepsilon)$ contient (r, ε) et $\delta(A)(r, z, \varepsilon)$ contient (q, ε). Donc $\delta(A)(p, yz, \varepsilon)$ contient (p, ε).

— $A_{p,p} \to \varepsilon$. Ce cas est exclus.

□

Proposition 89.

Si $\delta(A)(p, x, \varepsilon)$ contient (q, ε), $x \in \Sigma(A)^\star$, alors $A_{p,q} \to^* x$.

Démonstration. La preuve se fait par récurrence sur la longueur k du calcul

$k = 0 : x = \varepsilon$ et donc $p = q$. On applique $A_{p,q} \to \varepsilon$.

Supposons la propriété vraie pour tout calcul de longueur inférieure à k. Considérons un calcul de longueur $k + 1$.

— le symbole t n'est pas dépilé au cours du calcul : trivial.

— le symbole t est dépilé quand l'automate est dans l'état r. $\delta(A)(p, y, \varepsilon)$ contient (r, ε) et $\delta(A)(r, z, \varepsilon)$ contient (q, ε) avec $x = yz$. Ces calculs sont de longueur au plus k. Grâce à l'hypothèse de récurrence, on a $A_{p,r} \to^* y$ et $A_{r,q} \to^* s$, tel que $A_{p,q} \to A_{p,r}A_{r,q} \to^* yz = x$ d'où $A_{p,q} \to^* x$.

□

Ces deux propriétés réciproques prouvent le théorème. □

Corollaire 24.

Le problème de l'appartenance d'un mot à un langage de cette classe est décidable.

Démonstration. En effet, l'arrêt d'un automate à pile est décidable. □

26.4 Les grammaires régulières

Définition 102.

Une grammaire linéaire à gauche est une grammaire dont les règles sont de la forme

$$A \to Ba$$
$$A \to a$$

où $(A, B) \in (\mathcal{N})^2$, $a \in \mathcal{T}$.
Une grammaire linéaire à droite est une grammaire dont les règles sont de la forme

$$A \to aB$$
$$A \to a$$

avec les mêmes notations.
Une grammaire est dite régulière si elle est linéaire à gauche ou si elle est linéaire à droite.
Les règles sont simples, insensibles au contexte et d'une forme très restreinte. Il ne peut y avoir qu'un symbole non terminal au plus et il se trouve à une extrémité du mot.

Remarque 14.

On ne peut pas autoriser les deux types de règles simultanément dans une grammaire sans sortir de la classe des langages rationnels : on obtient les grammaires linéaires qui constituent une classe intermédiaire entre le type 2 et le type 3. Les règles d'une grammaire linéaire sont de la forme :

$$A \to aBb$$
$$A \to a$$

où $(A, B) \in (\mathcal{N})^2$, $(a, b) \in (\mathcal{T} \cup \{\varepsilon\})^2$.

Notation 41.

L'ensemble des langages engendrés par les grammaires linéaires est noté L_3.

Théorème 35.

$$L_3 = \mathbf{AF}$$

Démonstration. $L_3 \subseteq \mathbf{AF}$: Soit $\mathcal{G}$ une grammaire régulière et $u \in T(\mathcal{G})^\star$. On construit un automate fini A qui reconnait les mots de $\mathcal{G}$. On suppose la grammaire linéaire à droite.

Pour tout symbole non terminal B, on ajoute à l'automate un état q_B. Pour toute règle $B \to aB'$, on ajoute la transition $(q_B, a) \to q_{B'}$. D'autre part, pour toute transition $B \to a$, on ajoute la transition $(q_B, a) \to q_F$ où q_F est un état acceptant.

Ainsi, une dérivation d'un mot donne un chemin acceptant dans l'automate. On procède de même (en inversant les transition pour une grammaire linéaire à gauche.

$\mathbf{AF} \subseteq L_3$: Soit A un automate fini.

On construit, pour chaque état q, un symbole non terminal B_q. L'ensemble des symboles terminaux est l'alphabet de A.

Pour chaque transition $(q, a) \to q'$, on crée la règle $B_q \to aB_{q'}$ et on B_{q_0} comme axiome, où q_0 est l'état initial.

Ainsi, tout calcul acceptant se traduit par une dérivation dans la grammaire et réciproquement. □

Corollaire 25.

Le problème de l'appartenance d'un mot à un langage de cette classe est décidable.

Démonstration. En effet, l'arrêt d'un automate fini est certain. □

26.5 Propriétés

Proposition 90.

$$L_3 \subsetneq L_2 \subsetneq L_1 \subsetneq L_0 \subsetneq U$$

où U est l'univers de tous les langages.

Démonstration. Cette hiérarchie s'obtient simplement en comparant les modèles de calcul qui reconnaissent les différents type de langages. □

On donne des exemples de langages :

Exemple 23 (L_3)**.**

$a^\star b^\star$, $(aaab)^\star$, $\left\{a^{3i} \mid i > 0\right\}$

Exemple 24 ($L_2 \setminus L_3$)**.**

$\left\{a^i b^i \mid i > 0\right\}$, l'ensemble des palindromes, les langages de DYCK (ensemble des expressions bien parenthésées)

Exemple 25 ($L_1 \setminus L_2$)**.**

$\left\{a^i b^i c^i \mid i > 0\right\}$, $\left\{a^i b^k c^i d^k \mid i > 0, k > 0\right\}$, $\{uu \mid u \in \{a,b\}^\star\}$

26.6 Raffinement de la hiérarchie de CHOMSKY

La hiérarchie originale de CHOMSKY comprenait quatre classes mais on peut facilement en ajouter d'autres. On donne quelques exemples

— entre le type 0 et le type 1, les langages récursifs, qui sont acceptés par les machines de TURING qui s'arrêtent toujours,

— entre le type 1 et le type 2, les langages à grammaires indexées, définis par des grammaires plus générales que les grammaires contextuelles,

— entre le type 2 et le type 3, les langages algébriques déterministes, pour lesquels il existe une caractérisation par automate, mais pas par les grammaires.

Chapitre 27

Les machines de TURING rouillées

On étudie ici une restriction des machines de TURING. La thèse de CHURCH-TURING avance que les machines de TURING calculent tout ce qui est algorithmiquement faisable. Il est donc intéressant de voir ce qu'il en est pour des restrictions de ce modèle de calcul. Ici, on s'intéresse aux machines de TURING qui changent leur état un nombre borné de fois, quelle que soit l'entrée. On appelle ce modèle, les machines de TURING rouillées. Le but de ce chapitre est de caractériser sa puissance de calcul. On impose cette limitation car restreindre le nombre d'état n'est pas suffisant puisque la puissance de calcul ne change pas tant qu'il en reste au moins 2 [Sha57].

Le problème de l'arrêt pour les machines à un état est déjà étudié par G.T. HERMAN [Her69] et Y. SAOUTER [Sao95]. Seulement il n'existe aucun résultat sur les machines de TURING rouillées bien que certains aspects de ces modèles soient communs.

Par la suite, on donne des algorithmes pour reconnaitre certains langages : certaines expression rationnelles (étoile de KLEENE d'un mot et certains cas particuliers de $(\sum u_i)^\star$), l'arithmétique de PRESBURGER, les langages de DYCK à une parenthèse et un langage contextuel non algébrique. On montre aussi comment calculer la taille de l'entrée. On montre que le problème de l'arrêt de ces machines est décidable et on montre que son castor affairé est calculable et on en donne une majoration. Enfin, on prouve que ces machines calculent uniquement des fonctions de $\mathcal{E}^3$ de la hiérarchie de GRZEGORCZYK [Grz53] et toutes les fonctions de $\mathcal{E}^1$.

Ce modèle est un sujet de recherche en cours. Il n'est par conséquent que très mal cerné et il reste un grand nombre de question ouvertes. Certaines de ces questions sont données par la suite. Comme ce modèle et les notions connexes intéressent peu de monde, ces questions peuvent rester ouverte encore longtemps. Toute contribution est appréciée.

27.1 Définitions

On désigne par le terme de graphe de transition, le graphe dont les sommets sont les états de la machine de TURING considérée. Les arcs sont les transitions décrites par la fonction de transition. Les arcs sont donc étiquetés par deux symboles et un mouvement. Ici, on ne se servira pas de ces étiquettes.

On s'intéresse uniquement aux chemins qu'on peut faire dans ce graphe. En particulier, à la longueur des chemins après avoir simplifier les boucles.

Définition 103 (Machine de TURING rouillée)**.**

Soit M une machine de TURING. M est une machine de TURING rouillée si son graphe de transition est acyclique (aux boucles près).

Notation 42.

On note $\mathfrak{M}$ l'ensemble des machines de TURING rouillées.

Définition 104 (Transition charnière)**.**

Soit $M \in \mathbb{M}$. Soit $(q,a) \in Q \times \Gamma$, la transition $(q,a) \to (q',a',m) = \delta(q,a)$ est dite charnière si $q \neq q'$.

Notation 43.

Pour $M \in \mathbb{M}$ et un mot $x \in \Sigma^\star$, on note $\mathfrak{C}(M,x)$ le nombre de transitions charnières effectué par M sur l'entrée x. Si M ne s'arrête pas, $\mathfrak{C}(M,x)$ peut ne pas être défini.

Proposition 91.

$$\forall M \in \mathfrak{M}, \forall x \in \Sigma^\star, \mathfrak{C}(M,x) \leqslant |Q|$$

Notation 44 (Nombre d'oxydation)**.**

Pour $M \in \mathfrak{M}$, on note $\mathfrak{C}(M)$ le nombre maximal de transitions charnières qu'atteint M.

$$\mathfrak{C}(M) = \max\{\mathfrak{C}(M, x) \mid x \in \Sigma^\star\}$$

On l'appelle le nombre d'oxydation de M.

Proposition 92.

$$\forall M \in MTR, \mathfrak{C}(M) \text{ est défini}$$

Définition 105.

On dit que M est une machine de TURING rouillée au sens faible si son nombre d'oxydation est fini (même si son graphe de transition n'est pas acyclique).

Notation 45.

On note $\mathfrak{M}_f$ l'ensemble des machines de TURING rouillées au sens faible.

$$\mathfrak{M}_f = \{M \in \mathbb{M} \mid \exists c \in \mathbb{N} : \mathfrak{C}(M) \leqslant c\}$$

Proposition 93.

$$\forall M \in \mathfrak{M}_f, \mathfrak{C}(M) \text{ est défini}$$

Il est clair que pour toute machine de TURING rouillée au sens faible, il existe une machine de TURING rouillée qui fait le même calcul (avec la même complexité) et donc le graphe des transitions est acyclique (aux boucles près). Il suffit de dupliquer chaque état autant de fois que nécessaire qui est un nombre borné par le nombre d'oxydation.

On peut supposer que tout mot en entrée est encadré par des symboles particuliers μ_G à gauche et μ_D à droite. En effet, il est possible de positionner ces symboles en 4 changements d'états sans changer les autres transitions ni rajouter d'autres symboles.

Proposition 94.

$\mathfrak{M}$ est stable par la composition des machines de TURING.

On utilisera abondamment la composition pour enchainer plusieurs étapes dans un algorithme.

Proposition 95.

$\mathfrak{M}_f$ est stable par la composition des machines de TURING.

Proposition 96.

$$\mathfrak{M} \subsetneq \mathfrak{M}_f$$

Par conséquent, toute propriété vraie sur les machines de TURING rouillées au sens faible est vraie sur les machines de TURING rouillées.

Une fonction calculable par une machine de TURING rouillée est dite MTR-calculable.

27.2 Algorithmes, fonctions calculées et langages reconnus

27.2.1 Étoile de KLEENE d'un mot

$(ab)^\star$

Le principe est de parcourir le mot en faisant un pas en arrière à chaque b non parcouru. On marque les lettres par le nombre de fois qu'on les traverse. Ainsi, toute lettre doit être marquée exactement du nombre 2. On refuse si on parcourt une lettre déjà marquée d'un 2 ou si on trouve, à la fin, une lettre marquée par un 1 ou non marquée.

On construit la machine :

— $Q = \{q_0, q_1, q_Y, q_N\}$
— $\Gamma = \{B, \mu_G, \mu_D, a, (a,1), (a,2), b, (b,1), (b,2)\}$
— $\Sigma = \{a, b\}$
— $F = \{q_Y\}$

δ est décrit par :

état / symbole	q_0	q_1
a	$q_0, (a,1), \longrightarrow$	$q_N, a, \texttt{HALT}$
$(a,1)$	$q_0, (a,2), \longrightarrow$	$q_N, (a,1), \texttt{HALT}$
$(a,2)$	$q_N, (a,2), \texttt{HALT}$	$q_1, a, \longleftarrow$
b	$q_0, (b,1), \longleftarrow$	$q_N, b, \texttt{HALT}$
$(b,1)$	$q_0, (b,2), \longrightarrow$	$q_N, (b,1), \texttt{HALT}$
$(b,2)$	$q_N, (b,2), \texttt{HALT}$	$q_1, b, \longleftarrow$
μ_D	$q_1, \mu_D, \longleftarrow$	
μ_G	$q_N, \mu_G, \texttt{HALT}$	$q_Y, \mu_G, \texttt{HALT}$

Les transitions non précisées n'arrivent pas.

Théorème 36.

La machine précédente termine toujours et en 2 transitions charnières au plus. Elle termine dans q_Y si le mot en entrée est dans $(ab)^\star$ et termine dans q_N sinon.

On donne un exemple de l'exécution sur le mot initial *ababab*.

q_0 : $\ldots$ B B μ_G a b a b μ_D B B $\ldots$
q_0 : $\ldots$ B B μ_G $(a,1)$ b a b μ_D B B $\ldots$
q_0 : $\ldots$ B B μ_G $(a,1)$ $(b,1)$ a b μ_D B B $\ldots$
q_0 : $\ldots$ B B μ_G $(a,2)$ $(b,1)$ a b μ_D B B $\ldots$
q_0 : $\ldots$ B B μ_G $(a,2)$ $(b,2)$ a b μ_D B B $\ldots$
q_0 : $\ldots$ B B μ_G $(a,2)$ $(b,2)$ $(a,1)$ b μ_D B B $\ldots$
q_0 : $\ldots$ B B μ_G $(a,2)$ $(b,2)$ $(a,1)$ $(b,1)$ μ_D B B $\ldots$
q_0 : $\ldots$ B B μ_G $(a,2)$ $(b,2)$ $(a,2)$ $(b,1)$ μ_D B B $\ldots$
q_1 : $\ldots$ B B μ_G $\ldots$ $\ldots$ $\ldots$ $\ldots$ μ_D B B $\ldots$
q_1 : $\ldots$ B B μ_G a b a b μ_D B B $\ldots$
q_Y : $\ldots$ B B μ_G a b a b μ_D B B $\ldots$

On peut aussi donner un cas qui ne marche pas à cause de deux a consécutifs.

q_0 : $\ldots$ B B μ_G a b a a b μ_D B B $\ldots$
q_0 : $\ldots$ B B μ_G $(a,1)$ b a a b μ_D B B $\ldots$
q_0 : $\ldots$ B B μ_G $(a,1)$ $(b,1)$ a a b μ_D B B $\ldots$
q_0 : $\ldots$ B B μ_G $(a,2)$ $(b,1)$ a a b μ_D B B $\ldots$
q_0 : $\ldots$ B B μ_G $(a,2)$ $(b,2)$ a a b μ_D B B $\ldots$
q_0 : $\ldots$ B B μ_G $(a,2)$ $(b,2)$ $(a,1)$ a b μ_D B B $\ldots$
q_0 : $\ldots$ B B μ_G $(a,2)$ $(b,2)$ $(a,1)$ $(a,1)$ b μ_D B B $\ldots$
q_0 : $\ldots$ B B μ_G $(a,2)$ $(b,2)$ $(a,1)$ $(a,1)$ $(b,1)$ μ_D B B $\ldots$
q_0 : $\ldots$ B B μ_G $(a,2)$ $(b,2)$ $(a,1)$ $(a,2)$ $(b,1)$ μ_D B B $\ldots$
q_0 : $\ldots$ B B μ_G $(a,2)$ $(b,2)$ $(a,1)$ $(a,2)$ $(b,2)$ μ_D B B $\ldots$
q_1 : $\ldots$ B B μ_G $(a,2)$ $(b,2)$ $(a,1)$ $(a,2)$ $(b,2)$ μ_D B B $\ldots$
q_1 : $\ldots$ B B μ_G $(a,2)$ $(b,2)$ $(a,1)$ $(a,2)$ b μ_D B B $\ldots$
q_1 : $\ldots$ B B μ_G $(a,2)$ $(b,2)$ $(a,1)$ a b μ_D B B $\ldots$
q_N : $\ldots$ B B μ_G $(a,2)$ $(b,2)$ $(a,1)$ a b μ_D B B $\ldots$

Et avec un facteur bb.

q_0 : $\ldots$ B B μ_G a b b a μ_D B B $\ldots$
q_0 : $\ldots$ B B μ_G $(a,1)$ b b a μ_D B B $\ldots$
q_0 : $\ldots$ B B μ_G $(a,1)$ $(b,1)$ b a μ_D B B $\ldots$
q_0 : $\ldots$ B B μ_G $(a,2)$ $(b,1)$ b a μ_D B B $\ldots$
q_0 : $\ldots$ B B μ_G $(a,2)$ $(b,2)$ b a μ_D B B $\ldots$
q_0 : $\ldots$ B B μ_G $(a,2)$ $(b,2)$ $(b,1)$ a μ_D B B $\ldots$
q_N : $\ldots$ B B μ_G $(a,2)$ $(b,2)$ $(b,1)$ a μ_D B B $\ldots$

$(a^k)^\star$

On se donne $k \in \mathbb{N}^*$. Dans cet algorithme, le but est de parcourir le mot en faisant des aller-retours. Chaque lettre porte deux marques. La première est une marque G ou D qui indique si on doit parcourir la lettre vers la gauche ou vers la droite. Cette marque change à chaque fois. La seconde marque est un compteur qui compte le nombre de passages de la gauche vers la droite (modulo k). Ainsi, le mot est de la forme $(a^k)^n$ si et seulement si la première lettre a été parcourue un nombre de fois multiple de n.

— $Q = \{q_0, q_1, q_2, q_Y, q_N\}$
— $\Gamma = \{B, \mu_G, \mu_D, a\} \bigcup \{a\} \times [\![0, k-1]\!] \times \{D, G\}$

— $\Sigma = \{a\}$
— $F = \{q_Y\}$

symbole \ état	q_0	q_1
a	$q_0, (a, 0, D), \longleftarrow$	
(a, i, G) $\quad i < k-1$	$q_0, (a, i+1, D), \longleftarrow$	
$(a, k-1, G)$	$q_0, (a, 0, D), \longleftarrow$	
(a, i, D)	$q_0, (a, i, G), \longrightarrow$	
μ_G	$q_0, \mu_G, \longrightarrow$	$q_2, \mu_G, \longrightarrow$
μ_D	$q_1, \mu_D, \longleftarrow$	
x $\quad x \neq \mu_G$		$q_1, x, \longleftarrow$

symbole \ état	q_2
$(a, k-1, G)$	$q_Y, (a, k-1, G), \texttt{HALT}$
x $\quad x \neq (a, k-1, G)$	$q_N, x, \texttt{HALT}$

Proposition 97.

La machine précédente termine toujours et en 3 transitions charnières au plus. Elle termine dans q_Y si le mot en entrée est dans $(a^k)^\star$ et termine dans q_N sinon.

On donne un exemple d'exécution sur l'entrée a^6 pour $k = 3$.

q_0 :	B μ_G	a	a	a	a	μ_D B
q_0 :	B μ_G	$(a,0,D)$	a	a	a	μ_D B
q_0 :	B μ_G	$(a,0,D)$	a	a	a	μ_D B
q_0 :	B μ_G	$(a,0,G)$	a	a	a	μ_D B
q_0 :	B μ_G	$(a,0,G)$	$(a,0,D)$	a	a	μ_D B
q_0 :	B μ_G	$(a,1,D)$	$(a,0,D)$	a	a	μ_D B
q_0 :	B μ_G	$(a,1,D)$	$(a,0,D)$	a	a	μ_D B
q_0 :	B μ_G	$(a,1,G)$	$(a,0,D)$	a	a	μ_D B
q_0 :	B μ_G	$(a,1,G)$	$(a,0,G)$	a	a	μ_D B
q_0 :	B μ_G	$(a,1,G)$	$(a,0,G)$	$(a,0,D)$	a	μ_D B
q_0 :	B μ_G	$(a,1,G)$	$(a,1,D)$	$(a,0,D)$	a	μ_D B
q_0 :	B μ_G	$(a,2,D)$	$(a,1,D)$	$(a,0,D)$	a	μ_D B
q_0 :	B μ_G	$\ldots$	$\ldots$	$\ldots$	a	μ_D B

Suite sur la page suivante

$$
\begin{array}{lccccccc}
q_0: & \mathrm{B}\ \mu_G & (a,2,G) & (a,1,G) & (a,0,G) & a & \mu_D\ \mathrm{B} \\
q_0: & \mathrm{B}\ \mu_G & (a,2,G) & (a,1,G) & (a,0,G) & (a,0,D) & \mu_D\ \mathrm{B} \\
q_0: & \mathrm{B}\ \mu_G & (a,2,G) & (a,1,G) & (a,1,D) & (a,0,D) & \mu_D\ \mathrm{B} \\
q_0: & \mathrm{B}\ \mu_G & (a,2,G) & (a,2,D) & (a,1,D) & (a,0,D) & \mu_D\ \mathrm{B} \\
q_0: & \mathrm{B}\ \mu_G & (a,0,D) & (a,2,D) & (a,1,D) & (a,0,D) & \mu_D\ \mathrm{B} \\
q_0: & \mathrm{B}\ \mu_G & \dots & \dots & \dots & \dots & \mu_D\ \mathrm{B} \\
q_0: & \mathrm{B}\ \mu_G & (a,0,G) & (a,2,G) & (a,1,G) & (a,0,D) & \mu_D\ \mathrm{B} \\
q_0: & \mathrm{B}\ \mu_G & \dots & \dots & \dots & \dots & \mu_D\ \mathrm{B} \\
q_1: & \mathrm{B}\ \mu_G & (a,0,G) & (a,2,G) & (a,1,G) & (a,0,D) & \mu_D\ \mathrm{B} \\
q_2: & \mathrm{B}\mu_G & (a,0,G) & (a,2,G) & (a,1,G) & (a,0,D) & \mu_D\ \mathrm{B} \\
q_N: & \mathrm{B}\mu_G & (a,0,G) & (a,2,G) & (a,1,G) & (a,0,D) & \mu_D\ \mathrm{B}
\end{array}
$$

$u^\star$

On se donne un alphabet fini Σ et un mot $u \in \Sigma^\star$. On note $k = |u|$ et $u = u_0 \dots u_{k-1}$.

On cherche à reconnaitre les mots de la forme $u^\star$. L'idée est de numéroter chaque lettre avec le même procédé que pour $(a^k)^\star$. Ensuite on vérifie que le mot a une longueur multiple de k et que chaque lettre est bien placée.

Pour cela on prend une machine avec

— $Q = \{q_0, q_1, q_2, q_3, q_Y, q_N\}$

— $\Gamma = \{B, \mu_G, \mu_D\} \bigcup \Sigma \bigcup (\Sigma \times [\![0, k-1]\!] \times \{G, D\})$

— $F = \{q_Y\}$

Pour chaque lettre x de l'alphabet, la fonction de transition est la même que la précédente où on remplace a par x partout. On change en plus les q_Y en q_3 et on ajoute les règles suivantes :

symbole \ état		q_3
(a,i,D)	$a = u_i$	$q_3, a, \longrightarrow$
(a,i,D)	sinon	$q_N, (a,i,D), \texttt{HALT}$
μ_D		$q_Y, \mu_D, \texttt{HALT}$

Proposition 98.

La machine précédente termine toujours et en 4 transitions charnières au plus. Elle termine dans q_Y si le mot en entrée est dans $u^\star$ et termine dans q_N sinon.

27.2.2 $(u+v)^\star$

$|u| = |v|$

On se donne $n \in \mathbb{N}^*$, un alphabet fini Σ ainsi que deux mots $(u, v) \in (\Sigma^\star)^2$ tels que $|u| = |v| = n$.

Le but est de donner une méthode systématique de construction de machine de TURING rouillée pour reconnaitre $(u+v)^\star$.

On note $u = u(0) \dots u(n-1)$ et $v = v(0) \dots v(n-1)$. On appelle h le mot initialement sur le ruban et $l = |h|$.

Dans un premier temps, la machine numérote les lettres de h avec les entiers de $[\![0, n-1]\!]$. Si l n'est pas un multiple de n, on refuse l'entrée. Sinon, on obtient le mot $h(0)_{n-1}h(1)_{n-2} \dots h(n-1)_0 h(n)_{n-1} \dots h(l)_0$.

Ensuite, on commence à reconnaitre u. Dans l'état $q_{u,0}$, la machine parcours le mot de la gauche vers la droite. Si elle lit la lettre $u(n)_0$, on inscrit $u(n)'_0$ à la place et on fait un mouvement vers la gauche.

A l'issue de cette étape, les occurrences bien placées de la dernière lettre de u sont marquées par l'apostrophe et les lettres précédentes sont marquées par un M.

Par la suite, on poursuit en regardant si les lettres marquées par M sont bien placées pour u. On suppose qu'on en est à l'étape p. À l'issue de cette étape, toutes les $p+1$ dernières lettres bien placées de u sont marquées par l'apostrophe, les autres lettres restent dans leur état d'origine (avec la numérotation).

Après n étapes de ce type, tous les mots u bien placés seront donc entièrement marqués par l'apostrophe. Et on passe maintenant à la reconnaissance de v. À ce point, certaines lettres de v sont également marquées. Ces lettres sont celles du plus long suffixe commun à u et v. Il suffit donc de reprendre les mêmes tables que précédemment mais à partir de la dernière lettre qui n'appartient pas au plus long suffixe commun. Quand ces étapes sont terminées toutes les lettres de u et de v sont marquées par l'apostrophe. On refuse il reste une lettre non marquée à la fin de l'algorithme. Si toutes les lettres sont marquées alors le mot était bien dans $(u+v)^\star$ et la machine termine en acceptant.

Proposition 99.

La construction précédente fournit une machine de TURING rouillée qui termine sur toute entrée en au plus $4n+7$ transitions charnières. De plus, elle termine dans q_Y si le mot en entrée est dans $(u+v)^\star$ et dans q_N sinon.

symbole \ état	$q_{u,0}$	$q_{u,1}$
$u(n-1)_0$	$q_{u,0}, u(n-1)'_0, \longleftarrow$	
$a_i \quad a_i \neq u(n-1)_0$	$q_{u,0}, a_i^D, \longrightarrow$	
a_i^D	$q_{u,0}, a_i^M, \longrightarrow$	$q_{u,1}, a_i, \longleftarrow$
a_i^M		$q_{u,1}, a_i^M, \longleftarrow$
a'_i		$q_{u,1}, a'_i, \longleftarrow$
μ_D	$q_{u,1}, \mu_D, \longleftarrow$	
μ_G		$q_{u,2}, \mu_G, \longrightarrow$

symbole \ état	$q_{u,2p}$	$q_{u,2p+1}$
$u(n-1-p)_p$	$q_{u,2p}, u(n-1-p)'_p, \longleftarrow$	
$a_i \quad a_i \neq u(n-1-p)_p$	$q_{u,2p}, a_i^D, \longrightarrow$	
a_i^D	$q_{u,2p}, a_i^M, \longrightarrow$	$q_{u,2p+1}, a_i, \longleftarrow$
a_i^M		$q_{u,2p+1}, a_i^M, \longleftarrow$
a'_i		$q_{u,2p+1}, a'_i, \longleftarrow$
μ_D	$q_{u,2p+1}, \mu_D, \longleftarrow$	
μ_G		$q_{u,2(p+1)}, \mu_G, \longrightarrow$

$(\sum u_i)^\star$ **avec** $|u_i| = n$

On se donne $n \in \mathbb{N}^*$, un ensemble fini I, un alphabet fini Σ et une famille $(u_i) \in (\Sigma^\star)^I$ de mot sur Σ indexée par I telle que $\forall i \in I, |u_i| = n$.

On peut généraliser la construction précédente pour reconnaitre les mots de $\left(\sum_{i\in I} u_i\right)^\star$. Il suffit de procéder de la même façon et de commencer à reconnaitre un mot à partir du plus grand suffixe déjà reconnu pour les mots précédents.

$\left(a^{k_1} + b^{k_2}\right)^\star$

On se donne un alphabet de deux lettres $\Sigma = \{a, b\}$ et $(k_1, k_2) \in \mathbb{N}^{*2}$.

On cherche à construire une machine qui reconnait les mots de $\left(a^{k_1} + b^{k_2}\right)^\star$. Le but est de procéder en deux passes, une pour chaque lettre, en numérotant les lettres à la façon de l'algorithme pour reconnaitre $(a^k)^\star$. Pour la première passe, chaque a se comporte comme précédemment et chaque b ne fait qu'un déplacement à droite. Ainsi on numérote les séries de a consécutifs. Dans la seconde passe, on numérote les séries de b. Il suffit maintenant de regarder les premières lettres de chaque série. Encore une fois, on le fait en deux passes.

On parcourt le mot de la droite vers la gauche. On marque tous les a qu'on lit. Lorsqu'on lit un b numéroté avec 0, on le marque et on fait un pas en arrière. Ainsi on lit le premier a de la série qui suit. Si le a est numéroté avec $k_1 - 1$, on poursuit l'algorithme, sinon on refuse l'entrée. On procède de même lorsqu'on rencontre μ_G. On fait une seconde passe en échangeant les rôles de a et de b et on vérifie que les b sont bien numérotés par $k_2 - 1$. Si on finit la seconde passe sans rejeter l'entrée, alors le mot appartient bien à $\left(a^{k_1} + b^{k_2}\right)^\star$ et on accepte l'entrée.

La machine a les caractéristiques suivantes :

— $Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_Y, q_N\}$
— $\Sigma = \{a, b\}$
— $\Gamma = \left\{B, \mu_G, \mu'_G, \mu_D, a, b\right\} \bigcup \{a, b, a', b'\} \times [\![0, k-1]\!] \times \{G, D\}$
— $F = \{q_Y\}$

On donne les tables de transitions. Elles sont découpées de façon logique, par étapes, et sont accompagnées d'un commentaire afin de comprendre ce qui se passe.

On commence par numéroter les a.

symbole \ état	q_0	q_1
a	$q_0, (a, 0, D), \longleftarrow$	
$(a, i, G) \qquad i < k_1 - 1$	$q_0, (a, i+1, D), \longleftarrow$	
$(a, k_1 - 1, G)$	$q_0, (a, 0, D), \longleftarrow$	
(a, i, D)	$q_0, (a, i, G), \longrightarrow$	
b	$q_0, b, \longrightarrow$	
μ_G	$q_0, \mu_G, \longrightarrow$	$q_2, \mu_G, \longrightarrow$
μ_D	$q_1, \mu_D, \longleftarrow$	
$x \qquad x \neq \mu_G$		$q_1, x, \longleftarrow$

Ensuite, on numérote les b.

symbole \ état	q_2
b	$q_2, (b, 0, D), \longleftarrow$
$(b, i, G) \qquad i < k_2 - 1$	$q_2, (b, i+1, D), \longleftarrow$
$(b, k_2 - 1, G)$	$q_2, (b, 0, D), \longleftarrow$
(b, i, D)	$q_2, (b, i, G), \longrightarrow$
a	$q_2, a, \longrightarrow$
(a, i, D)	$q_2, (a, i, D), \longrightarrow$
(a, i, G)	$q_2, (a, i, G), \longrightarrow$
μ_G	$q_2, \mu_G, \longrightarrow$
μ_D	$q_3, \mu_D, \longleftarrow$

On vérifie la longueur des séquences de a.

symbole \ état	q_3	q_4
(a,i,G)	$q_3,(a',0,G),\longleftarrow$	
$(b,0,G)$	$q_3,(b',0,G),\longrightarrow$	
(a',k_1-1,G)	$q_3,(a',k_1-1,G),\longleftarrow$	$q_4,(a,k_1-1,G),\longrightarrow$
$(a',i,G)\quad i<k_1-1$	$q_N,(a',i,G),\texttt{HALT}$	$q_4,(a,i,G),\longrightarrow$
$(b',0,G)$	$q_3,(b',0,G),\longleftarrow$	$q_4,(b,0,G),\longrightarrow$
$(b,i,G)\quad i>0$	$q_3,(b,i,G),\longleftarrow$	$q_4,(b,i,G),\longrightarrow$
μ_G	$q_3,\mu'_G,\longrightarrow$	
μ'_G	$q_4,\mu_G,\longrightarrow$	
μ_D		$q_5,\mu_D,\longleftarrow$

Et on termine en vérifiant la longueur des séquences de b.

symbole \ état	q_5
(b,i,G)	$q_5,(b',0,G),\longleftarrow$
$(a,0,G)$	$q_5,(a',0,G),\longrightarrow$
(b',k_2-1,G)	$q_5,(b',k_2-1,G),\longleftarrow$
$(b',i,G)\qquad i<k_2-1$	$q_N,(b',i,G),\texttt{HALT}$
$(a',0,G)$	$q_5,(a',0,G),\longleftarrow$
$(a,i,G)\qquad i>0$	$q_5,(a,i,G),\longleftarrow$
μ_G	$q_5,\mu'_G,\longrightarrow$
μ'_G	$q_Y,\mu_G,\texttt{HALT}$

Proposition 100.

La machine précédente termine toujours et en 6 transitions charnières au plus. Elle termine dans q_Y si le mot en entrée est dans $(a^{k_1}+b^{k_2})^\star$ et termine dans q_N sinon.

$\left(\sum a_i^{k_i}\right)^\star$

On se donne un ensemble fini I, un alphabet $\Sigma=\{a_i\}_{i\in I}$ indexé par I, une famille $(k_i)\in\mathbb{N}^I$ de naturels indexé par I.

En généralisant la méthode précédente, on peut reconnaitre tout mot de $\left(\sum_{i\in I} a_i^{k_i}\right)^\star$. Le nombre d'états dépend directement et affinement de $|I|$.

Il suffit de faire $|I|$ numérotations (pour chacune des lettres, en ignorant les autres comme précédemment) puis faire $|I|$ vérification de la numérotation. Chaque étape est très similaires à celles de la machine précédente.

27.2.3 L'arithmétique de PRESBURGER

Définitions

On se donne un ensemble fini I, un alphabet fini $\Sigma_0 = \{x_i\}_{i\in I}$ indexé par I.

Définition 106.

Les prédicats de l'arithmétique de PRESBURGER sont :
- $\sum_{i\in I} a_i|u|_{x_i} \geqslant k$,
- $\sum_{i\in I} a_i|u|_{x_i} \equiv b[k]$,
- une combinaison booléenne de prédicats de l'arithmétique de PRESBURGER.

où $(a_i) \in \mathbb{Z}^I$, $(b,k) \in \mathbb{Z}^2$.
Les prédicats de l'arithmétique de PRESBURGER sont des prédicats à une place dont la variable est un mot de $\Sigma_0^\star$.

Dans la suite, dans une expression arithmétique de l'arithmétique de PRESBURGER, on écrira x_i pour $|u|_{x_i}$. Cette notation ne cause pas d'ambiguïté car les prédicats de l'arithmétique de PRESBURGER n'ont qu'une variable.

$\Sigma a_i x_i \equiv b[k]$

Cette classe de prédicats se calcule en utilisant le principe du calcul de $(a^k)^\star$. Si $\forall i \in I, a_i = 1$, le calcul est précisément identique. Pour adapter, il suffit de passer a_i fois sur chaque occurrence de x_i.

Comme on travaille modulo k, on peut considérer que $\forall i \in I, a_i \geqslant 0$.
- $Q = \{q_0, q_1, q_2, q_3, q_N, q_Y\}$
- $\Sigma = \Sigma_0$
- $\Gamma = \{B, 0_G, 0_D\} \bigcup \Sigma \bigcup [\![0, k-1]\!] \bigcup \{L_i \mid i \in [\![0, k-1]\!]\}$
- $F = \{q_Y\}$

état / symbole	q_0	q_1
$x_i \quad a_i > 0$	$q_1, L_{a_i-1}, \longrightarrow$	$q_1, a_i, \longrightarrow$
$x_i \quad a_i = 0$	$q_1, L_{k-1}, \longrightarrow$	$q_1, 0_G, \longrightarrow$
μ_D		$q_1, \mu_D, \longleftarrow$
i		$q_1, i, \longleftarrow$
$L_i \quad i < k-1$		$q_2, L_{i+1}, \longrightarrow$
L_{k-1}		$q_2, L_0, \longrightarrow$

symbole \ état	q_2	q_3
μ_D	$q_3, \mu_D, \longleftarrow$	
$L_i \quad i < k-1$	$q_2, L_{i+1}, \longrightarrow$	
L_{k-1}	$q_2, L_0, \longrightarrow$	
0_G	$q_2, 0_D, \longleftarrow$	$q_3, 0_D, \longleftarrow$
0_D	$q_2, 0_G, \longrightarrow$	
$i \quad i > i$	$q_2, 0_D, \longleftarrow$	
L_b		$q_Y, L_b, \texttt{HALT}$
$L_i \quad i \neq b$		$q_N, L_i, \texttt{HALT}$

$\Sigma a_i x_i \geqslant b$

On commence par écrire $\sum_{a_i>0} a_i x_i$ en unaire à côté du mot. Pour cela, on écrit pour chaque lettre la paire $(\max(a_i, 0), \min(0, a_i))$ puis on transfert par aller-retours chacun de ces chiffres des premières composantes à l'extrémité du ruban. On rencontre un chiffre i non nul, on écrit $i-1$ et on repars dans l'autre sens. Lorsqu'on rencontre le symbole blanc, on écrit un $(1, 0)$ et on fait demi tour. Pour cela, il faut considérer μ_D comme B.

Cette fois, on écrit $\sum_{a_i<0} a_i x_i$ à côté du mot en unaire dans la seconde composante de la paire précédente. Il suffit de procéder exactement de la même façon en considérant la seconde composante des paires.

Ensuite, il suffit de parcourir ces nombres en unaire et de finir quand le premier finit. Ainsi, on sait que si le second finit avant le premier, $\sum a_i x_i \geqslant 0$. Il suffit alors d'ajuster la variable b. Pour ce faire, il suffit de remplir avec b $(1, 0)$ ou $(0, 1)$ à l'extrémité du mot selon que b est positif ou négatif.

Combinaisons booléennes

On va justifier qu'on peut faire les combinaisons booléennes des prédicats précédents en esquissant un raisonnement par induction.

On peut se restreindre aux cas où on n'a que les connecteurs $\neg$, $\wedge$ et $\vee$.

Dans le cas de $\neg$, il suffit d'échanger F_A et F_R.

Si on a des machines M et N pour reconnaitre respectivement P et Q. Pour reconnaitre $P \vee Q$, il suffit alors de rediriger les états de F_R de M vers l'état initial de N en prenant soin d'intercaler quelques états pour restaurer le ruban d'origine grâce à la méthode des paires.

Dans le cas de $P \wedge Q$, il suffit de rediriger les états de F_A de M vers l'état initial de N en ayant restauré le ruban.

Proposition 101.

Pour tout formule de l'arithmétique de PRESBURGER, il existe une machine de TURING rouillée qui termine toujours et qui termine dans un état acceptant si le mot en entrée vérifie la formule et finit dans un état rejetant sinon.

27.2.4 Taille de l'entrée

On se donne $b \in \mathbb{N} \setminus \{0, 1\}$ et Σ un alphabet fini.

On va ici décrire une machine qui écrit à côté du mot en entrée la taille de celui-ci en base b. Par conséquent on n'inscrit pas μ_G à gauche du mot.

Ici, on veut une machine faite pour le calcul et non pour la décision. Aussi il n'y a pas d'état rejetant mais seulement des états acceptant ne servant que d'états finaux pour terminer le calcul.

La machine fait des aller-retours sur le mot en rayant un symbole à chaque fois. À l'extrémité du mot, on met un compteur qui est incrémenté à chaque fois que la tête de lecture entre dans celui-ci. Après l'incrémentation, la tête de lecture en ressort sans faire de changement d'état. Pour faire cela, on utilise le plus simple algorithme d'incrémentation : on parcourt le compteur en partant des chiffres de poids faible en changeant tous les $b-1$ en 0 et en augmentant le premier chiffre inférieur à $b-1$. Après avoir augmenté ce chiffre, on repart vers les bits de poids faible sans rien changer afin de ressortir du compteur. Si on dépasse le compteur (car il représentait un nombre de la forme $b^k - 1$), on fixe le premier blanc rencontré à 1 et on repart vers les bits de poids faible. Lorsqu'on rencontre μ_D, tous les symboles du mots ont été rayés donc on a incrémenté le compteur un nombre de fois égale à la longueur du mot. On termine alors le calcul et le compteur contient la taille de l'entrée.

Proposition 102.

Pour tout $b \in \mathbb{N} \setminus \{0, 1\}$, il existe une machine de TURING rouillée qui termine toujours et écrit la taille du mot en entrée en base b.

27.2.5 Le langage de DYCK

On veut reconnaitre les langages de DYCK (ie. les mots bien parenthésés) avec un seul type de parenthèses.

On recherche et raye les parenthèses correspondantes dans le mot en entrée.

Tant qu'on trouve une parenthèse ouvrante, on la marque et on se déplace à droite. Si on trouve une parenthèse fermante, on écrit D et on repart à gauche. Lorsqu'on trouve une parenthèse ouvrante marquée, on la remplace par G et on se déplace à droite. Lorsqu'on trouve un G, on le remplace par un D et on se déplace à gauche. Au contraire, si on trouve un D, on écrit G et on se déplace à droite. A la fin de cette étape, si on trouve μ_D on passe dans l'état suivant, sinon on trouve μ_G et on refuse l'entrée. Dans cette seconde étape, on parcourt le mot de droite à gauche. Si on ne trouve que des G avant μ_G, on accepte. Si on rencontre tout autre caractère, l'entrée est rejetée.

Proposition 103.

Il existe une machine de TURING rouillée qui termine toujours en au plus 3 transitions charnières et qui accepte l'entrée si et seulement si elle appartient au langage de DYCK avec un type de parenthèses.

— $Q = \{q_0, q_1, q_Y, q_N\}$
— $\Gamma = \{B, \mu_G, \mu_D, D, G, (,), (')\}$
— $\Sigma = \{(,)\}$
— $F = \{q_Y\}$

état / symbole	q_0	q_1
$($	$q_0, ('$, $\longrightarrow$	$q_N, (,$ HALT
$)$	$q_0, D, \longleftarrow$	$q_N,),$ HALT
$('$	$q_0, G, \longrightarrow$	$q_N, ('$, HALT
D	$q_0, G, \longrightarrow$	$q_N, D,$ HALT
G	$q_0, D, \longleftarrow$	$q_1, G, \longleftarrow$
μ_G	$q_N, \mu_G,$ HALT	$q_Y, \mu_G,$ HALT
μ_D	$q_1, \mu_D, \longleftarrow$	

27.2.6 $|u|_a = 2^{|u|_b}$

On veut reconnaitre les mots u sur $\{a, b\}$ tels que $|u|_a = 2^{|u|_b}$.

Pour ce faire, on compte successivement les a et les b et on inscrit ces totaux à gauche du mot en utilisant des couples de chiffres : le premier représente le nombre de a et le second le nombre de b.

Le principe est d'écrire en binaire le nombre de a, en unaire le nombre de b et de comparer les longueurs. Le nombre de a doit être une puissance de 2

donc il doit comporter t 0 précédés par un 1. Le nombre de b doit comporter t symboles. A la fin de l'exécution, le ruban doit être du type

$\ldots$	B	B	$(1,0)$	$(0,1)$	$\ldots$	$(0,1)$	u_0	$\ldots$	$u_{\lvert u\rvert-1}$	B	B	$\ldots$

Les nombres sont écrits de la droite vers la gauche pour pouvoir utiliser des compteurs comme précédemment.

Cette machine a les caractéristiques suivantes.

— $Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_Y, q_N\}$
— $\Sigma = \{a, b\}$
— $\Gamma = \Sigma \bigcup \{\mu_G, \mu_G, B, 0, 0', 1, a', b', (0,1), (1,1), (0,1'), (1,1')\} \bigcup \Sigma \times \{G, D\}$
— $F = \{q_Y\}$

On donne les tables de transitions. Comme précédemment, on a découpé ces tables et on commente chaque sous partie.

On commence par faire les numérotations.

état / symbole	q_0	q_1	q_2
a	$q_0, a', \longleftarrow$		
a'	$q_0, (a, G), \longrightarrow$		
(a, G)	$q_0, (a, D), \longrightarrow$	$q1, a, \longleftarrow$	
(a, D)	$q_0, (a, G), \longrightarrow$		
b	$q_0, (b, G), \longrightarrow$		$q_2, b', \longleftarrow$
b'			$q_2, (b, G), \longrightarrow$
(b, G)	$q_0, (b, D), \longleftarrow$	$q1, b, \longleftarrow$	$q_2, (b, D), \longleftarrow$
(b, D)	$q_0, (b, G), \longrightarrow$		$q_2, (b, G), \longrightarrow$
μ_D	$q_1, \mu_D, \longleftarrow$		$q_3, \mu_D, \longleftarrow$
μ_G	$q_0, 1, \longrightarrow$		$q_2, (0,1), \longrightarrow$
B	$q_0, 1, \longrightarrow$		$q_2, (0,1), \longrightarrow$
0	$q_0, 1, \longrightarrow$	$q_2, 0, \longrightarrow$	$q_2, (0,1), \longrightarrow$
1	$q_0, 0', \longleftarrow$	$q_2, 1, \longrightarrow$	$q_2, (1,1), \longrightarrow$
$0'$	$q_0, 0, \longrightarrow$		
$(0,1)$			$q_2, (0,1'), \longleftarrow$
$(0,1')$			$q_2, (0,1), \longrightarrow$
$(1,1)$			$q_2, (1,1'), \longleftarrow$
$(1,1')$			$q_2, (1,1), \longrightarrow$

Puis on vérifie les longueurs relatives des nombres inscrits.

symbole \ état	q_3	q_4	q_5
(a, G)	$q_3, a, \longleftarrow$		
(b, G)	$q_3, b, \longleftarrow$		
$(0, 1)$	$q_4, (0, 1), \longleftarrow$		
1	$q_5, 1, \longleftarrow$	$q_5, 1, \longleftarrow$	$q_N, 1, \texttt{HALT}$
$(0, 1)$		$q_4, (0, 1), \longleftarrow$	
$(1, 1)$		$q_N, (1, 1), \texttt{HALT}$	
B			$q_Y, 0, \texttt{HALT}$
0			$q_N, 0, \texttt{HALT}$

Proposition 104.

La machine précédente termine toujours et en 6 transitions charnières au plus. Elle termine dans q_Y si le mot u en entrée vérifie $|u|_a = 2^{|u|_b}$ et termine dans q_N sinon.

On en déduit directement deux autres résultats.

Théorème 37.

On peut reconnaitre des langages qui ne vérifient pas l'arithmétique de PRESBURGER.

Lemme 14 ([Car08c]).

Tout langage rationnel ou algébrique vérifie l'arithmétique de PRESBURGER

Théorème 38.

On peut reconnaitre des langages qui ne sont pas algébriques.

27.3 Calculabilité

27.3.1 Les machines de TURING rouillées à plusieurs rubans

Nous avons considéré jusqu'ici que des machines de TURING rouillées avec un seul ruban. Dans cette sous-section, nous nous intéressons maintenant à la puissance de calcul des machines de TURING rouillées avec au moins 2 rubans. Par la suite les machines considérées auront de nouveau qu'un seul ruban.

Dans ce modèle, les rubans et les mouvements des têtes de lecture/écriture sur chacun sont indépendants. L'état est global et est soumis à la même restriction que précédemment.

Théorème 39.

Les machines de TURING rouillées avec au moins deux rubans sont équivalentes aux machines de TURING.

Démonstration. Les machines de TURING rouillées ne sont qu'une restriction des machines de TURING aussi, elles ne peuvent pas être plus puissantes.

Réciproquement, on prend $\mathcal{U} = (Q_\mathcal{U}, \Gamma_\mathcal{U}, B_\mathcal{U}, \Sigma_\mathcal{U}, q_{0\mathcal{U}}, \delta_\mathcal{U}, F_{A\mathcal{U}}, F_{R\mathcal{U}})$ une machine de TURING universelle. On construit une machine de TURING rouillée à deux rubans $\mathcal{R} = (Q, \Gamma, B, \Sigma, q_0, \delta, F_A, F_R)$ qui simule $\mathcal{U}$.

On a $Q = \{q_0, q_Y, q_N\}$. On ajoute des transitions qui permettent de simuler les transitions de $\mathcal{U}$ en stockant l'état de $\mathcal{U}$ sur un ruban de $\mathcal{R}$ et le ruban de $\mathcal{U}$ sur l'autre ruban de $\mathcal{R}$.

Pour toute transition $(q, a) \to (q', a', m)$ de $\mathcal{U}$ on ajoute à $\mathcal{R}$ la transition :
— si $q' \in F_R : (q_0, (q, a)) \to (q_N, (q', a'), (0, m))$
— si $q' \in F_A : (q_0, (q, a)) \to (q_Y, (q', a'), (0, m))$
— si $q' \in Q \setminus (F_A \cup F_R) : (q_0, (q, a)) \to (q_0, (q', a'), (0, m))$

On prend ainsi $\Gamma = Q_\mathcal{U} \bigcup \Gamma_\mathcal{U}$, $\Sigma = \Sigma_\mathcal{U}$, $B = B_\mathcal{U}$, $F_A = \{q_Y\}$ et $F_R = \{q_N\}$.

Ainsi le premier ruban de $\mathcal{R}$ contient l'état de $\mathcal{U}$ et le second est le ruban de $\mathcal{U}$. Aussi, on a bien la même puissance de calcul et par conséquent, $\mathcal{R}$ est universelle. Enfin, $\mathcal{R}$ est effectivement rouillée car elle ne peut faire que deux changement d'état au plus : $q_0 \to q_Y$ et $q_0 \to q_N$. □

27.3.2 Décidabilité de l'arrêt

On va prouver que l'arrêt des machines de TURING rouillées est décidable.

Lemme 15 ([Sao95]).

L'arrêt d'une machine de TURING à un état est décidable.

On peut ainsi déterminer dans quel état une telle machine termine si on sait qu'elle termine.

Théorème 40.

Le problème de l'arrêt pour les machines de TURING rouillées est décidable.

Démonstration. Il suffit de décider le changement d'état pour chaque transition. Il faut faire $\mathfrak{C}(M, x)$ décisions de ce type. On finit par trouver un état terminal ou un état qui ne termine pas. Dans tous les cas, on peut décider l'arrêt d'une machine de TURING rouillée par machine de TURING. □

Comme l'arrêt est décidable, il est évident que les machines de TURING rouillées constituent un modèle de calcul qui n'est pas TURING-complet.

Remarque 15.

On peut maintenir la même preuve avec une machine de TURING rouillée au sens faible.

Il suffit d'appliquer la même méthode. On ne sait pas le nombre de décision qu'il y aura à faire mais comme c'est nécessairement un nombre fini (puisqu'on a une machine de TURING rouillée au sens faible), le calcul terminera et cette généralisation de l'arrêt reste décidable.

Corollaire 26.

Pour toute machine $M \in \mathfrak{M}_f$, la fonction

$$\begin{aligned} \Sigma^\star &\to \mathbb{N} \\ x &\mapsto \mathfrak{C}(M, x) \end{aligned}$$

est calculable.

27.3.3 Le castor affairé

La démonstration de la décidabilité de l'arrêt d'une machine de TURING à un état montre qu'une telle machine écrit au plus $|\Gamma|^{|x|}$ symboles où Γ est l'alphabet de la machine et x le mot en entrée.

De la même façon, en un changement d'état, on majore par $|\Gamma|^{|x|}$ le nombre de symboles qu'on peut écrire sur le ruban. On peut le minorer par $(|\Gamma| - 3)^{|x|}$ puisqu'il suffit pour atteindre cette borne de considérer le mot en entrée comme un nombre en base Σ et de le réécrire en unaire. L'algorithme demande 3 symboles supplémentaires, d'où $|\Sigma| + 3 = |\Gamma|$.

Ainsi, on peut majorer le castor affairé des machines de TURING rouillées. Il s'agit de la fonction qui donne le nombre maximum de symbole qu'on peut écrire avec un nombre donné de symboles, d'état et avec un mot en entrée d'une longueur donnée.

Pour une machine à $|Q|$ états, on passe donc au plus par $|Q|$ états. On note $\mathfrak{BB}(|Q|, |\Sigma|, |x|)$ le nombre maximum de symboles qu'on peut écrire par une machine de TURING rouillée à $|Q|$ états, $|\Sigma|$ symboles et avec un mot de longueur $|x|$ initialement sur le ruban. Aussi par une récurrence évidente, on a $\mathfrak{BB}(|Q|, |\Sigma|, |x|) \leqslant |\Sigma|^{|\Sigma|^{\cdot^{\cdot^{|\Sigma|^{|x|}}}}}$ avec $|Q|$ exposants $|\Sigma|$. On peut aussi majorer brutalement par $\mathfrak{BB}(|Q|, |\Sigma|, |x|) \leqslant (|\Sigma| + |x|) \uparrow\uparrow |Q|$. Néanmoins un castor affairé normal part d'un ruban vide. On peut donc supposer que les n premiers états servent à écrire un mot initial, puis que les $m = |Q| - n$ suivant servent au calcul. On a alors un castor affairé majoré par $|\Sigma|^{|\Sigma|^{|\Sigma|^{\cdot^{\cdot^{|\Sigma|^n}}}}}$ avec m exposants $|\Sigma|$. Cette fonction est clairement maximisée pour $n = 0$. On définit donc $\mathfrak{BB}(|Q|, |\Sigma|)$ qui est le nombre de symboles qu'on peut écrire au maximum depuis un ruban vide. On peut en déduire une majoration $\mathfrak{BB}(|Q|, |\Sigma|) \leqslant |\Sigma|^{|\Sigma|^{|\Sigma|^{\cdot^{\cdot^{|\Sigma|}}}}}$ avec $|Q| - 1$ exposants $|\Sigma|$ puis $\mathfrak{BB}(|Q|, |\Sigma|) \leqslant |\Sigma| \uparrow\uparrow (|Q| - 1) \leqslant |\Sigma| \uparrow\uparrow |Q|$, ce qui est une majoration primitive récursive. D'autre part, on peut donner une minoration du même ordre.

On peut ainsi donner une majoration du castor affairé.

Théorème 41.

$$\mathfrak{BB}(|Q|, |\Sigma|) \leqslant \Sigma \uparrow\uparrow (|Q| - 1)$$

27.3.4 Classe des fonctions calculées

Des fonctions incalculables

Premièrement, il y a des fonctions qu'on ne calcule pas puisque l'arrêt est décidable. On peut par exemple expliciter une de ces fonctions : $x \mapsto 2^{2^{x\uparrow\uparrow x}}$. En effet, l'écriture de cette fonction prendra un espace $c2^{x\uparrow\uparrow x}$ pour une constante c qui dépend de $|\Sigma|$. Comme cette fonction dépasse le castor affairé, on ne peut pas écrire le résultat de cette fonction en dehors d'un nombre fini de valeur. Cette fonction est donc MTR-incalculable alors qu'elle est calculable par machine de TURING.

Les fonctions primitives récursives

Maintenant, caractérisons les fonctions calculées. Dans un premier temps, on peut montrer que toutes les fonctions calculées par des machines de TURING rouillées sont primitives récursives. En effet, la fonction permettant d'initialiser la machine est primitive récursive, ainsi que la fonction qui permet de réaliser une transition. Aussi il suffit de détecter l'arrêt, ce qui peut se faire avec une minimisation bornée. Il suffit que la borne soit primitive récursive. Or, on a montré que l'arrêt ne peut pas intervenir après avoir écrit dans plus de $|\Sigma|^{|x|}$ cases, soit, au plus, après $|\Sigma|^{|\Sigma|^{|x|}}$ transitions. Aussi, on peut coder le calcul de chaque machine de TURING rouillées par une fonction primitive récursive puisque les fonctions primitives récursives sont stables par minimisation bornée [Ros84].

Théorème 42.

Les machines de TURING rouillées ne calculent que des fonctions primitives récursives.

Ce théorème est relativement puissant. Il exclus la majorité des fonctions calculables pour être calculables par machine de TURING rouillées.

Raffinement grâce à la hiérarchie de GRZEGORCZYK

On veut maintenant avoir un encadrement plus précis des fonctions que calculent les machines de TURING rouillées. On utilise pour cela la hiérarchie de GRZEGORCZYK [Grz53].

Théorème 43 ([Ros84]).

Toute fonction primitive récursive majorée par une fonction de $\mathcal{E}^3$ appartient à $\mathcal{E}^3$.

Les fonctions calculées par machine de TURING rouillées sont majorée par une exponentielle itérée de son castor affairé qui est elle même une itérée de l'exponentielle. Ainsi, toutes ces fonctions sont majorées par une fonction de la classe $\mathcal{E}^3$ de la hiérarchie de GRZEGORCZYK. Or, ces fonctions sont primitives récursives et majorées par une fonction de $\mathcal{E}^3$, par conséquent, elles sont toutes dans $\mathcal{E}^3$.

Théorème 44.

Les machines de TURING rouillées ne calculent que des fonctions de $\mathcal{E}^3$.

D'autre part, on peut essayer de minorer la classe des fonctions calculées.

Proposition 105.

Soit $f \in \mathcal{E}^1$, il existe des entiers naturels $(a_i)_{i\in[\![0,n]\!]}$ tels que

$$f(x_1, \ldots, x_n) = a_0 + \sum_{i=1}^{n} a_i x_i$$

Démonstration. On fait la preuve par induction. On sait que les fonctions de bases (projections, 0, successeur, somme) sont affines.

On a deux schéma de construction : la composition et la récursion limitée [Ros84].

Il est évident que l'ensemble des fonctions affines est stable par composition.

On se donne deux fonctions affines g et h. On construit $f = Rec(g, h)$ par récursion limitée avec comme borne, une fonction j elle-même affine.

On rappelle la construction récursive bornée :

$$\begin{aligned} f(x_1,\dots,x_n,0) &= g(x_1,\dots,x_n) \\ f(x_1,\dots,x_n,n+1) &= h(x_1,\dots,x_n,n,f(x_1,\dots,x_n,n)) \\ f(x_1,\dots,x_n,y) &\leqslant j(x_1,\dots,x_n,y) \end{aligned}$$

On pose $g(x_1,\dots,x_n) = g_0 + \sum_{i=1}^{n} g_i x_i$ et $h(x_1,\dots,x_n,k,y) = h_0 + \sum_{i=1}^{n} h_i x_i + ak + by$.

On peut aisément montrer par récurrence

$$f(x_1,\dots,x_n,k) = b^k \left(g_0 + \sum_{j=1}^{n} x_j g_j \right) + \sum_{i=0}^{k-1} b^i \left(h_0 + \sum_{j=1}^{n} x_j h_j \right) + a \sum_{i=1}^{k-1} i b^{k-i-1}$$

Si $b > 1$ on a des termes exponentiels (sauf si la famille des h_i, la famille des g_i et a sont nuls), ce qui ne peut pas être majoré par j qui est une fonction affine. Ainsi, on a nécessairement $b \leqslant 1$. On a alors que f est une fonction affine en chacune de ses variables.

□

Théorème 45.

Les machines de TURING rouillées calculent toutes les fonctions de $\mathcal{E}^1$.

On peut en effet calculer la somme en unaire et c'est la seule chose requise pour calculer les fonctions de $\mathcal{E}^1$.

Il reste cependant plusieurs questions non résolues. On peut espérer caractériser plus finement les fonctions calculées ou déterminer si on peut reconnaître tous les langages rationnels ou non et plus généralement, trouver la classe des langages reconnus. Il reste en particulier à savoir si $(u+v)^\star$ est reconnaissable.

On peut aussi tenter d'étendre les résultats de calculabilité aux machines de TURING rouillées au sens faible. Cela repose surtout sur la décidabilité du nombre de transition charnière de ces machines. Cela revient à trouver f calculable tel que $f(|Q|, |\Sigma|)$ majore le nombre d'oxydation $\mathfrak{C}(M)$.

On peut aussi considérer des extensions de ce modèle. On peut donner une version non déterministe de ces machines. L'autre point important d'évolution consiste à majorer le nombre de transition charnière non par une constante mais par une fonction de la taille de l'entrée. On peut ainsi avoir une notion de complexité en nombre de changement d'états.

Bibliographie

[Car08a] O. Carton. Langages formels, calculabilité et complexité, pages 115–119. École Normale Supérieure, Juin 2008. Machines de TURING.

[Car08b] O. Carton. Langages formels, calculabilité et complexité, pages 111–158. École Normale Supérieure, Juin 2008. Calculabilité.

[Car08c] O. Carton. Langages formels, calculabilité et complexité, volume 101. Vuibert, 2008.

[Grz53] A. Grzegorczyk. Some classes of recursive functions. rozprawy matematyczne no. 4. instytut matematyczny polskiej akademii nauk, warschau 1953, 46 s. 1953.

[Her69] G. Herman. The uniform halting problem for generalized one-state turing machines. Information and Control, 15(4) :353–367, 1969.

[Knu92] D. Knuth. Two notes on notation. The american mathematical monthly, Mai 1992. `arXiv:math/9205211v1`.

[Ren01] P. Rendell. TURING universality of the game of life. AA book P.Rendell chapter draft 3, Novembre 2001.

[Ren11a] P. Rendell. A TURING machine in CONWAY's game of life, extendable to a universal TURING machine, Décembre 2011. `http://rendell-attic.org/gol/tm.htm`.

[Ren11b] P. Rendell. A universal TURING machine in CONWAY's game of life. AA book P.Rendell chapter draft 3, Novembre 2011.

[Rig07] M. Rigo. Algorithmique et calculabilité. Fonction d'ACKERMANN. Université de Liège, `http://www.discmath.ulg.ac.be/`, 2007.

[Rig09] M. Rigo. Algorithmique et calculabilité. Université de Liège, 2009.

[Ros84] H. Rose. Subrecursion : functions and hierarchies. Oxford Science Publications, 1984.

[RTH+12] T. Rokicki, A. Trevorrow, T. Hutton, D. Greene, J. Summers, M. Verver, and R. Munafo. Golly version 2.4, 2012. `http://golly.sourceforge.net/`.

[Sao95] Y. Saouter. Halting problem for one-state turing machines. 1995.

[Sha57] C. Shannon. A universal TURING machine with two internal states. Automata studies, 34 :157–165, 1957.

[Tur36] A. Turing. On computable numbers, with an application to the Entscheidungsproblem. Novembre 1936.