Leçon 913 : Machines de Turing. Applications.

Julie Parreaux

1 Introduction

En 1900, le mathématicien allemand David Hilbert énonce vingt-trois problèmes considérés comme les problèmes mathématiques les plus importants du XX$^{\text{ième}}$ siècle. Le dixième problème [1] de cette liste est à l'origine de la théorie de la calculabilité qui a été développée par les mathématiciens pour y répondre. Cette théorie cherche à répondre à la question *qu'est-ce qui est calculable ?* Pour cela, les mathématiciens ont cherché à modéliser le processus du calcul via des modèles formels qui sont à la base de cette théorie. Ils permettent d'appréhender les limites non-physiques mais bien conceptuelles du calcul et de l'informatique en général. Ils modélisent formellement la notion de calcul effectif ou encore celle d'algorithme.

Les premiers modèles de calcul introduits répondant à la question *qu'est-ce qui est calculable ?* sont le λ-calcul et les fonctions μ-récursives qui modélisent formellement les fonctions mathématiques récursives. Les fonctions μ-récursives sont les plus proches dans leur syntaxe de ces fonctions récursives. Le λ-calcul quant-à lui est plus proche de la sémantique des langages fonctionnels : c'est le modèle abstrait utilisé pour étudier ces langages. Ils ont été introduits dans les années 1930 afin de conforter la thèse d'Alonzo Church qui définit les limites de la calculabilité. Ils permettent de décrire pour la première fois la frontière entre les fonctions calculables ou non en les classifiant [1]. Cependant le modèle du λ-calcul ne permet pas de classifier les fonctions calculables : elles sont toutes considérées comme équivalentes. Le modèle des fonctions μ-récursive fait une première distinction dans les fonctions calculables : celles définies sans une minimisation non-bornée et les autres. Cela établit un début de hiérarchie dans les fonctions calculables.

Afin d'approfondir les travaux sur les modèles de calculs formels et leur compréhension, Alan Turing introduit en 1936 les machines qui porte son nom [6]. Cette machine s'inscrit dans la suite des travaux sur les modèles de calcul formels. Cependant, Alan Turing réussit à prendre un nouveau point de vue sur le problème. En s'abstrayant d'un point de vue uniquement fonctionnel et très mathématique des modèles précédents, il imagine sa machine en répondant à la simple question *comment un humain calcule ?* Alan Turing remarque alors qu'un humain calcule avec un papier et un crayon en n'effectuant que des petits pas de calcul à chaque étape. Les machines de Turing sont alors une réponse d'un logicien à l'action du calcul réalisée par un humain. Elles introduisent la notion de langages (analogue à un humain qui calcule en utilisant un langage spécifique : le langage des mathématiques) décidables ou indécidables.

Les machines de Turing viennent à leur tour conforter la théorie de la calculabilité via la thèse de Church. Les langages décidables par les machines de Turing sont bien entendus équivalents aux fonctions calculables qui sont définies par les modèles précédents. Elles n'apportent pas de nouvelles fonctions calculables considérées comme non-calculable par les modèles précédents : ces modèles ont le même pouvoir d'expressivité. Par leur approche nova-

1. Le dixième problème de Hilbert s'interroge sur l'existence d'un algorithme (ou méthode de calcul effective) pour résoudre des équations diophantiennes de degrés quelconque.

trice, les machines de Turing sont, également, à l'origine de la théorie de la complexité. Elles permettent de classer un langage décidable en répondant à la question *quelles sont les ressources nécessaires pour le décider ?*, ce que les précédents modèles n'arrivaient pas à faire. Elles classent alors les langages décidables en fonction de la quantité de ressources (spatiales ou temporelles) utilisées par la machine de Turing qui les décident.

Les apports des machines de Turing dans les théories de la calculabilité et de la complexité sont des raisons essentielles justifiant leurs études. Cependant, ce ne sont pas les seuls apports. Par leur approche mécanique, elles sont aujourd'hui considérées comme le modèle abstrait des ordinateurs actuels. Elles englobent l'idée de procédure effective et donnent un moyen pratique de la mettre en place grâce à une machine.

Cette leçon se découpe en trois parties. On commence par définir le modèle de calcul qui est décrit par les machines de Turing (section 2). On continue en justifiant la thèse de Church et sa conséquence sur la théorie de la décidabilité (section 3). On termine par la théorie de la complexité introduite par les machines de Turing (section 4).

2 Les Machines de Turing : un modèle de calcul formel

Les machines de Turing sont un modèle de calcul formel introduit pour modéliser le procédé de calcul lorsqu'il est réalisé par un humain. Dans cette section, après avoir défini les modèles de calcul formels, nous allons introduire le vocabulaire nécessaire à la manipulation d'une machine de Turing. De plus, les machines de Turing peuvent calculer des fonctions en les évaluant comme le faisait les modèles de calculs précédents.

Définition 1 (Modèle de calcul [3])**.** Un *modèle de calcul* est un système qui associe à une entrée x, une sortie y en un nombre fini d'opérations élémentaires.

2.1 Vocabulaire autour des machines de Turing

Intuitivement, une machine de Turing est un automate fini muni d'un ruban, fini à gauche et infini à droite, sur lequel on peut lire et écrire sans aucune restriction. Notre objectif est de définir les langages acceptés et décidés par une machine de Turing qui sont les notions au centre de cette leçon. En effet, elles sont au cœur des théories de la calculabilité et de la complexité et donnent de nombreuses applications aux machines de Turing.

Définition 2 (Machine de Turing déterministe [7])**.** Une *machine de Turing déterministe* est décrite par un heptuplet $\mathcal{M} = (\mathcal{Q}, \Gamma, \Sigma, \delta, s, \#, F)$ où $\mathcal{Q}$ est l'ensemble fini d'états de la machine ; Γ est l'alphabet de travail qui est celui qu'on utilise sur le ruban ; $\Sigma \subseteq \Gamma$ est l'alphabet d'entrée ; $\delta : \mathcal{Q} \times \Gamma \to \mathcal{Q} \times \Gamma \times \{\leftarrow, \rightarrow\}$ est la fonction de transition de la machine ; $s \in \mathcal{Q}$ est l'état initial de la machine de Turing ; $\# \in \Gamma \setminus \Sigma$ est le symbole blanc qui est un symbole frais et $F \subseteq \mathcal{Q}$ est l'ensemble des états acceptants de la machine.

Les machines de Turing sont un modèle très expressif : elles sont au sommet de la hiérarchie de Chomsky [2]. Cependant en restreignant les actions que l'on peut réaliser sur le ruban, comme par exemple, la lecture seule ou l'utilisation d'un ruban de taille bornée, on peut définir des modèles moins expressifs bien connus de la théorie des automates.

2. La hiérarchie de Chomsky catégorise les automates et leurs familles de langages associés en fonction de leur expressivité. On retrouve à leur sommet les machines de Turing associées aux langages généraux, puis nous avons les automates linéairement bornés associés aux langages contractuel. Ensuite, nous avons les automates à pile non-déterministe associés aux langages algébriques. Enfin, les automates finis associés aux langages rationnels sont à la base de cette hiérarchie.

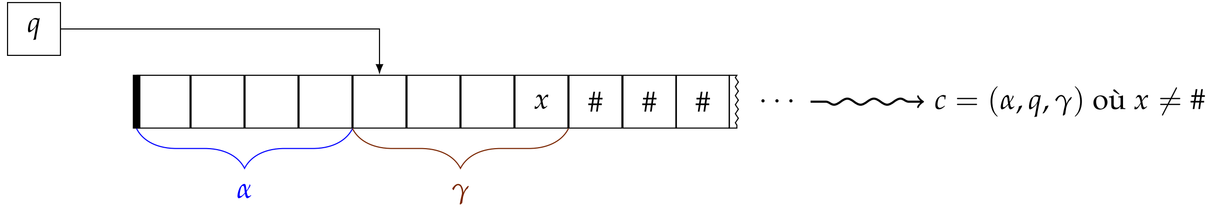

FIGURE 1 – Une configuration c pour une machine de Turing déterministe dans un état q avec le mot $\alpha\gamma$ écrit sur le ruban et telle que sa tête de lecture est sur la première lettre du mot γ.

Application 3. Un automate fini [3] est une machine de Turing déterministe parcourant une seule fois son ruban sans rien écrire : c'est une machine de Turing déterministe sans mémoire.

Pour décrire l'exécution d'une machine de Turing déterministe, nous devons décrire son évolution en fonction de son état courant et du mot inscrit sur son ruban. Nous utilisons la notion de configuration et une suite de ces configurations permet de décrire l'exécution d'une machine de Turing déterministe.

Définition 4 (Configuration [7])**.** Une *configuration* d'une machine de Turing déterministe $\mathcal{M} = (\mathcal{Q}, \Gamma, \Sigma, \delta, s, \#, F)$ est un élément de $(\Gamma^* \times \mathcal{Q} \times (\epsilon \cup \Gamma^* (\Gamma \setminus \{\#\})))$.

Cette définition de configuration ne fait pas apparaître dans la configuration l'infinité de symboles blancs complétant le ruban à droite : elle s'arrête au dernier symbole non blanc sur le ruban. Cette suite infinie de symboles blancs existe toujours quelques soit la configuration considérée et sa présence ne caractérise pas la configuration. De plus, dans cette définition, la position de la tête de lecture est donnée implicitement par la longueur de la première composante correspondant au mot inscrit sur le ruban avant la tête de lecture (voir Figure 1). Pour décrire une exécution d'une machine de Turing déterministe, il nous faut définir un moyen de passer d'une configuration à la suivante.

Définition 5 (Configuration suivante [7])**.** Soit $c = (\alpha a, q, b\gamma)$ une configuration dans la machine de Turing déterministe $\mathcal{M} = (\mathcal{Q}, \Gamma, \Sigma, \delta, s, \#, F)$. On note c' la *configuration suivante* de c que l'on définit comme suit.

— Si $\delta(q, b) = (q', b', \rightarrow)$, alors $c' = (\alpha a b', q, \gamma)$.
— Si $\delta(q, b) = (q', b', \leftarrow)$, alors $c' = (\alpha, q, a b' \gamma)$.

On note alors $c \vdash_{\mathcal{M}} c'$ le passage d'une configuration c à sa configuration suivante c' dans la machine de Turing déterministe $\mathcal{M}$, ou $c \vdash c'$ s'il n'y a pas d'ambiguïté sur la machine de Turing déterministe utilisée. De plus, on note $c \vdash^*_{\mathcal{M}} c'$ ou $c \vdash^* c'$ si on passe d'une configuration c à c' en plusieurs étapes.

Définition 6 (Exécution d'une machine de Turing déterministe [7])**.** Une *exécution* d'une machine de Turing déterministe $\mathcal{M} = (\mathcal{Q}, \Gamma, \Sigma, \delta, s, \#, F)$ sur un mot w est la suite de configuration, $(\epsilon, s, w) \vdash_{\mathcal{M}} c_1 \vdash_{\mathcal{M}} \ldots$, issue de la configuration initiale (ϵ, s, w) et maximale telle qu'elle soit infini, ou se terminant sur un état acceptant, ou sur une configuration dont la configuration suivante n'est pas définie.

La notion d'exécution d'une machine de Turing déterministe nous permet de définir les langages acceptés et décidés qui se distinguent en fonction de l'existence ou non d'une exécution infinie de la machine de Turing déterministe.

Définition 7 (Langage accepté et langage décidé [7])**.** Le *langage accepté* par une machine de Turing déterministe $\mathcal{M} = (\mathcal{Q}, \Gamma, \Sigma, \delta, s, \#, F)$, noté $L(\mathcal{M})$, est l'ensemble des mots w tels que $(\epsilon, s, w) \vdash^* (\alpha, q_f, \gamma)$ où $q_f \in F$.

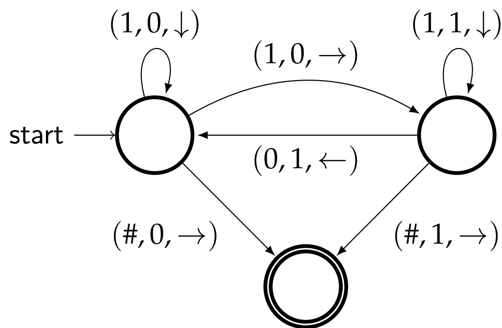

FIGURE 2 – Machine de Turing déterministe modélisant une multiplication par deux d'un nombre binaire.

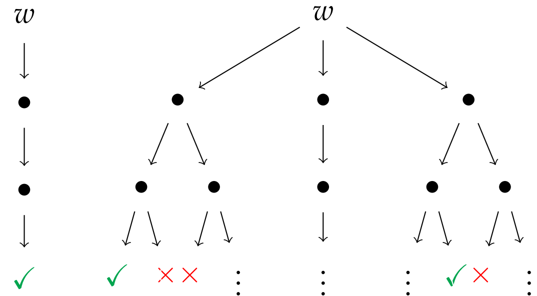

FIGURE 3 – Différence d'exécution entre une machine de Turing déterministe (à gauche) et d'une machine de Turing non-déterministe (à droite).

On dit que le langage $L(\mathcal{M})$ est *décidé* si $\mathcal{M}$ n'a pas d'exécution infinie.

Exemple 8. Le langage $L = a^n b^n c^n$ est décidé par une machine de Turing déterministe.

2.2 Les machines de Turing calculent

Les machines de Turing déterministes sont des modèles de calcul considérés comme les modèles abstraits des ordinateurs actuels. Ce sont des procédés mécaniques qui réalisent des calculs très généraux : elles permettent d'aller plus loin dans les calculs que les opérations arithmétiques de bases qu'elles maîtrisent. Elles permettent d'évaluer une fonction en une valeur. Cette notion fait le lien entre la théorie des fonctions calculables et celle des langages décidables. Nous allons alors introduire la notion de fonction calculable.

Définition 9 (Fonction calculable par une machine de Turing [7])**.** Une machine de Turing déterministe *calcule une fonction* $f : \Sigma^* \to \Sigma^*$ si, pour tout mot d'entrée w, elle s'arrête toujours dans une configuration où $f(w)$ se trouve sur le ruban.

Exemple 10 (Multiplication par deux [5]). Une machine de Turing déterministe peut effectuer la multiplication par deux d'un nombre binaire. On représente cette machine sur la Figure 2 : sur les arcs de son automate fini, on inscrit la fonction de transition sous la forme : (a, b, c) où a est la lettre lue sur le ruban, b est la lettre écrite à la place de a et $c \in \{\leftarrow; \downarrow; \rightarrow\}$.

3 Les machines de Turing confortent de la thèse de Church

Un des principaux apports des machines de Turing est la consolidation de la théorie introduite par la thèse de Church (1935) : la théorie de la calculabilité ou encore appelée théorie de la décidabilité (voir la sous-section 3.3). Un des aspects de cette théorie est que cette thèse à son origine peut en réalité s'énoncer de plusieurs manières selon le modèle de calcul que l'on considère sans en modifier la théorie. Autrement dit, pour tous les modèles de calculs aussi expressifs que les machines de Turing (comme les fonctions récursives ou le λ-calcul), on peut énoncer une variante équivalente de la thèse de Church.

Dans cette leçon, on travaille avec la thèse de Church exprimée pour le modèle de machines de Turing (également appelée thèse de Turing-Church) : *les fonctions calculables par un algorithme sont les fonctions calculables par une machine de Turing* [7]. On énonce une thèse : ce n'est ni

une hypothèse, ni un théorème. Nous ne pouvons pas la montrer formellement car la notion d'algorithme ne possède pas de définition formelle. Cependant nous allons mettre en avant deux arguments qui conforte cette thèse :

— le modèle de calcul des machines de Turing déterministe est robuste (voir la sous-section 3.1) ;

— les autres modèles de calculs formels tels que les fonctions μ-récursives et le λ-calcul sont équivalents aux machines de Turing déterministes (voir la sous-section 3.2).

3.1 Les machines de Turing sont un modèle robuste

Le modèle des machines de Turing est robuste : il n'existe pas d'extension naturelle d'une machine de Turing déterministe qui permette de décider un nouveau langage qui n'était pas encore décidable. Toute extension d'une machine de Turing est équivalente à une machine de Turing déterministe telle que l'on l'a défini à dans la définition 2. Dans cette leçon, nous nous sommes concentré sur trois exemples d'extensions [7] : les machines de Turing déterministe à rubans multiples, les machines de Turing déterministes à ruban bi-infini et les machines de Turing non-déterministes. On va montrer que chacune de ces extensions est équivalente à une machine de Turing déterministe au sens de leur expressivité, c'est-à-dire que les familles des langages qu'elles décident sont égales. Autrement dit, nous pouvons en simuler une avec l'autre et réciproquement.

Étude des machines de Turing déterministes à plusieurs rubans Une machine de Turing déterministe à plusieurs rubans est définie comme une machine de Turing déterministe comportant plusieurs rubans avec plusieurs têtes de lectures qui agissent simultanément mais indépendamment sur chacun des rubans. La fonction de transition prend en compte toutes les informations sur l'ensemble des rubans pour faire son choix et agit (en écriture ou en lecture) sur tous les rubans. Formellement on la définit par le même heptuplet $\mathcal{M} = (\mathcal{Q}, \Gamma, \Sigma, \delta, s, \#, F)$ de la machine de Turing déterministe (définition 2) tel que on redéfinit uniquement la fonction de transition par $\delta : \mathcal{Q} \times \Gamma^k \to \mathcal{Q} \times \Gamma^k \times \{\leftarrow; \rightarrow\}^k$ où $k \geq 1$. Une configuration d'une telle machine de Turing est un élément de $(\Gamma^*)^k \times \mathcal{Q} \times (\epsilon \cup \Gamma^*(\Gamma \setminus \{\#\}))^k$.

Les machines de Turing déterministes et les machines de Turing déterministes à plusieurs rubans sont équivalentes. Comme, une machine de Turing déterministe est une machine de Turing déterministe à plusieurs rubans, on montre la réciproque. On construit alors une machine de Turing déterministe simulant une machine de Turing déterministe à k rubans. Cette simulation consiste à définir comment utiliser l'unique ruban de notre machine de Turing déterministe. Sur ce dernier, on considère que les cases à la position i modulo k sur le ruban de la machine de Turing déterministe représentent les cases du ruban i de la machine de Turing que l'on simule. La figure 4 présente un exemple de cette simulation pour une machine de Turing déterministe à deux rubans.

Étude des machines de Turing déterministes à ruban bi-infini Une machine de Turing déterministe à ruban bi-infini est définie comme une machine de Turing déterministe dont on peut parcourir le ruban dans les deux sens. La principale différence entre une machine de Turing déterministe à ruban bi-infinie et une machine de Turing déterministe est dans la définition de la fonction transition : il n'y a aucune contrainte sur le déplacement de la tête de lecture sur le ruban. Formellement, on la définit par le heptuplet $\mathcal{M} = (\mathcal{Q}, \Gamma, \Sigma, \delta, s, \#, F)$ équivalent à la définition 2, sur lequel on élimine les contraintes définies sur la fonction δ. Lorsqu'on définit une configuration, on remarque qu'on trouve également une infinité de

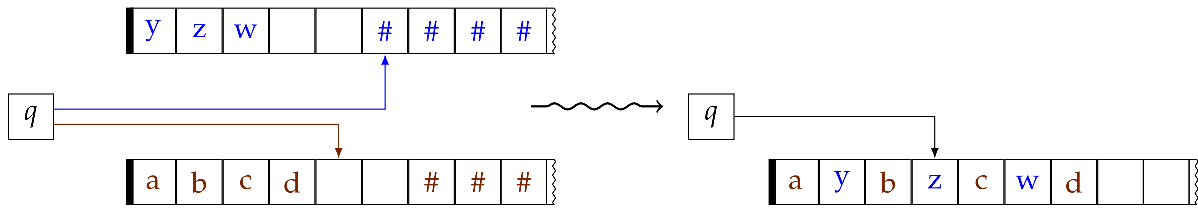

FIGURE 4 – Exemple d'une simulation d'une machine de Turing déterministe à deux rubans par une machine de Turing déterministe.

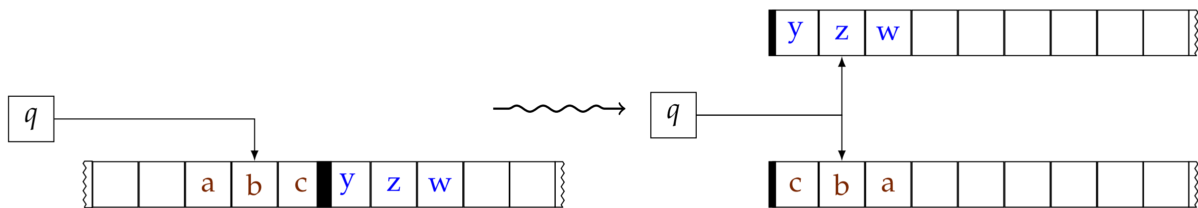

FIGURE 5 – Exemple d'une simulation d'une machine de Turing déterministe à ruban bi-infini par une machine de Turing déterministe à deux rubans avec une seule tête de lecture (elle lit une paire de symboles).

symboles blancs sur la partie gauche du ruban. Une configuration est alors un élément de $((\epsilon \cup (\Gamma \setminus \{\#\})\Gamma^*) \times \mathcal{Q} \times (\epsilon \cup \Gamma^*(\Gamma \setminus \{\#\}))$.

Les machines de Turing déterministes et les machines de Turing déterministes à ruban bi-infini sont équivalentes. Une machine de Turing déterministe est une machine de Turing déterministe à ruban bi-infini qui n'utilise pas la moitié de son ruban. Pour justifier l'équivalence entre une machine de Turing déterministe à un ruban et une machine de Turing bi-infini, on simule cette dernière par une machine à deux rubans avec une seule tête de lecture (les deux têtes de lectures de la machine de Turing déterministe à rubans multiples sont synchrones). Autrement dit on considère une machine avec un unique ruban de travail sur lequel on utilise une paire de symboles dans chacune des cases de son ruban. On fixe une case d'indice 0 sur le ruban bi-infini (la case comportant le symbole y dans la figure 5 représentant cette simulation). On construit deux rubans : le premier représente la partie droite du ruban bi-infini et le second la partie gauche. Pour se ramener à une machine de Turing déterministe, nous pouvons considérer l'interprétation avec la paire de symboles ou appliquer la simulation précédente.

Étude des machines de Turing non-déterministes Les machines de Turing non-déterministes introduisent des choix intrinsèques à celle-ci dans sa fonction de transition. Elles ont été introduites pour simplifier la définition de ces machines de Turing déterministes qui décident certains problèmes comme la satisfiabilité d'une formule du langage propositionnel. La différence notable entre une machine de Turing non-déterministe et une machine de Turing déterministe est la fonction de transition qui devient une relation de transition afin d'exprimer le choix de la machine de Turing non-déterministe. Formellement, on la définit alors à l'aide d'un heptuplet $\mathcal{M} = (\mathcal{Q}, \Gamma, \Sigma, \delta, s, \#, F)$ équivalent à celui de la machine de Turing déterministe où seule la fonction de transition est redéfinie en une relation : $\delta \subseteq (\mathcal{Q} \times \Gamma) \times (\mathcal{Q} \times \Gamma \times \{\leftarrow; \rightarrow\})$. Pour le machines de Turing non-déterministes, les configurations se définissent comme pour les machines de Turing déterministes.

Les transitions d'une machine de Turing non-déterministe sont définies par une relation donc pour une configuration donnée plusieurs configurations suivantes sont admissibles. La notion de configuration suivante doit être redéfinie afin de prendre en compte le choix que

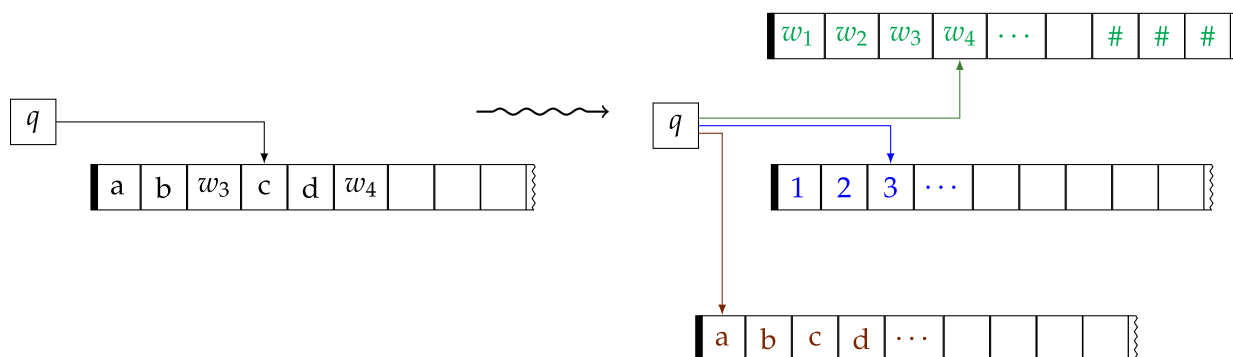

FIGURE 6 – Exemple d'une simulation d'une machine de Turing non-déterministe par une machine de Turing déterministe à trois rubans : le premier ruban (en vert) contient le mot d'entrée $w = w_1w_2w_3\dots$ qui est en lecture seule, le deuxième ruban (en bleu) contient les choix de la machine de Turing non-déterministes et le dernier ruban (en marron) est un ruban de travail.

l'on effectue. La notion d'un langage accepté évolue suite à cette définition. La configuration suivante transcrit le choix de la machine de Turing non-déterministe car elle est donnée parmi les configurations atteignables via la relation. Une suite de configurations (donc une exécution d'une machine de Turing non-déterministe) pour une entrée donnée acceptée positivement peut conduire à un rejet ou à une exécution infinie alors qu'une autre exécution donnerait une réponse positive (voir la Figure 3 pour un exemple d'exécution entre une machine de Turing non-déterministe et une machine de Turing déterministe). Par exemple, une machine de Turing non-déterministe vérifiant la satisfiabilité d'une formule du calcul propositionnel peut effectuer un mauvais choix et deviner une valuation ne satisfaisant pas la formule qui est pourtant satisfiable. On dit alors que un mot w est accepté s'il existe une suite de configurations (de choix) définissant une exécution acceptante. Le langage accepté par une machine de Turing non-déterministe est alors l'ensemble des mots acceptés par celle-ci.

Les machines de Turing non-déterministes sont équivalentes aux machines déterministes. En remarquant qu'une machine de Turing déterministe est en réalité une machine de Turing non-déterministe (une fonction est en réalité une relation particulière), il nous reste plus qu'à montrer la réciproque. On peut alors simuler une machine de Turing non-déterministe par une machine de Turing déterministe à trois rubans (voir Figure 6). Nous simulons celle-ci par une machine de Turing déterministe (voir Figure 4). Il ne nous reste plus qu'à définir la machine de Turing déterministe à trois rubans. Le premier ruban de celle-ci contient l'entrée en lecture seule que l'on peut parcourir comme on le souhaite. Le second contient la liste des différentes suites des choix possibles résolvant le non-déterministe (on sait qu'elle est finie). Le troisième simule la machine non-déterministe en exécutant les choix indiqués par le second ruban.

3.2 Les autres modèles de calcul décident les mêmes langages

Le modèle des machines de Turing possède la même expressivité que les modèles de calculs tels que les fonctions récursives ou le $\lambda-$calcul. Nous allons montrer que les fonctions récursives (plus précisément les fonctions μ-récursives) et que le λ-calcul sont équivalents aux machines de Turing en termes d'expressivité.

Étude des fonctions μ-récursives Nous définissons la syntaxe et la sémantique des fonctions μ-récursives [7]. Nous prouverons également leur équivalence avec les machines de Turing

déterministes.

Définition 11 (Syntaxe des expressions des fonctions μ-récursives)**.** Le langage $\mathcal{L}_{\mu R}$ des *expressions des fonctions μ-récursive* est défini par induction :

- $\mathbb{O}$ est une expression d'arité 0 ;
- σ est une expression d'arité 1 ;
- π_i^n où $n \in \mathbb{N}$ et $i \in \{1, \ldots, n\}$ est une expression d'arité n ;
- Si F est une expression d'arité n et que $G_1, \ldots, G_n$ sont des expressions d'arité l, alors $\circ(F, G_1, \ldots, G_n)$ est une expression d'arité l ;
- Si F est une expression d'arité n et que G est une expression d'arité $n+2$, alors $\text{rec}(F, G)$ est une expression d'arité $n+1$;
- Si F est une expression d'arité $n+1$, alors $\text{mu}(F)$ est une expression d'arité n.

Maintenant que nous avons une syntaxe pour notre langage, nous allons pouvoir lui en donner un sens : nous allons alors définir sa sémantique.

Définition 12 (Sémantique des expressions des fonctions μ-récursives)**.** Notons $\mathcal{F}_{\text{partielle}}(\mathbb{N}^k, \mathbb{N})$ l'ensemble des fonctions partielles de $\mathbb{N}^k$ dans $\mathbb{N}$. Avec les notations précédentes, on définit la fonction $[\![\ ]\!] : \mathcal{L}_{\mu R} \to \cup_{k \in \mathbb{N}} \mathcal{F}_{\text{partielle}}(\mathbb{N}^k, \mathbb{N})$ par induction :

- $[\![\mathbb{O}]\!] : \backslash \mapsto 0$;
- $[\![\sigma]\!] : x \mapsto x+1$;
- $[\![\pi_i^n]\!] : (x_1, \ldots, x_n) \mapsto x_i$;
- $[\![\circ(F, G_1, \ldots, G_n)]\!] : \overrightarrow{x} \mapsto [\![F]\!]\left([\![G_1]\!](\overrightarrow{x}), \ldots, [\![G_n]\!](\overrightarrow{x})\right)$;
- $[\![\text{rec}(F, G)]\!] = g$ où

$$\begin{array}{rccl} g : & \mathbb{N}^{n+1} & \to & \mathbb{N} \\ & (\overrightarrow{x}) & \mapsto & \left\{ \begin{array}{ll} [\![F]\!](\overrightarrow{x}) & \text{si } k = 0 \\ [\![G]\!]\left(\overrightarrow{x}, k-1, g(\overrightarrow{x}, k-1)\right) & \text{sinon} \end{array} \right. \end{array}$$

- $[\![\text{mu}(F)]\!] : \overrightarrow{x} \mapsto \mu_i.[\![F]\!](\overrightarrow{x}, i)$ où $\mu_i.[\![F]\!](\overrightarrow{x}, i)$ est le plus petit $i \in \mathbb{N}$ tel que $[\![F]\!](\overrightarrow{x}, i) \neq 0$ et est non défini si un tel i n'existe pas.

Définition 13. Une fonction $f : \mathbb{N}^k \to \mathbb{N}$ est une *fonction μ-récursive* si une expression de $\mathcal{L}_{\mu R}$ la dénote.

Exemple 14. Les fonctions μ-récursives permettent de définir des fonctions comme l'addition entre deux entiers naturels.

$$\text{plus}(n_1, n_2) = \left\{ \begin{array}{ll} \pi_1^2(n_1, n_2) & \text{si } n_2 = 0 \\ \sigma\left(\pi_3^3(n_1, n_2 - 1, \text{plus}(\mathrm{n}_1, \mathrm{n}_2 - 1))\right) & \text{sinon} \end{array} \right.$$

Les fonctions μ-récursives sont le premier modèle caractérisant les fonctions calculables des fonctions non-calculables. Les machines de Turing ont la même expressivité que celles-ci : les fonctions calculables par des machines de Turing sont également calculables par les fonctions μ-récursive et réciproquement.

Théorème 15. *Les fonctions μ-récursives sont exactement celles calculable par une machine de Turing.*

Pour montrer le théorème, nous avons besoin de coder des entiers en mots et vise-versa de manière à ce que ces opérations soient uniques (une chaîne donne un unique chiffre et ce chiffre donne de manière unique la même chaîne). Pour ce faire, nous utilisons la méthode de Gödel qui voit un mot comme un chiffre écrit dans une base précise. Nous avons à notre disposition deux fonctions gd qui permet de coder une chaîne de caractère à l'aide d'un entier et gd^{-1} sa réciproque (pour plus de détail voir section A). La preuve de ce théorème est donnée par les deux propositions suivantes qui donnent les deux inclusions.

Proposition 16. *Toute fonction calculable par une machine de Turing est une fonction μ-récursive.*

Démonstration. Soit M une machine de Turing sur un alphabet Σ calculant une fonction $f_{\mathcal{M}} : \Sigma^* \to \Sigma^*$. Quitte à déterminiser $\mathcal{M}$, on peut supposer que $\mathcal{M}$ est déterministe. On pose une fonction $gd : \Sigma^* \to \mathbb{N}$ codant les mots de Σ^* par des entiers. Nous allons montrer qu'il existe f une fonction μ-récursive tel que $f_{\mathcal{M}}(w) = gd^{-1}(f(gd(w)))$.

Hypothèses sur la machine $\mathcal{M}$ On peut supposer sans perte de généralité qu'il existe $m \in \mathbb{N}^*$ tel que $\mathcal{Q} = \{0; \dots ; m-1\}$ où $\mathcal{Q}$ est l'ensemble des états de $\mathcal{M}$. De plus, on suppose que $\mathcal{M}$ possède un unique état initial $q_{init} = 0$ et un unique état final $q_{final} = m-1$ qui est atteint lorsque $\mathcal{M}$ inscrit le dernier caractère de la solution (celui-ci est alors avant la tête de lecture sur le ruban). De plus, on suppose $\mathcal{M}$ complète de telle sorte que la fonction de transition $\delta : \mathcal{Q} \times \Sigma \to \mathcal{Q} \times \Gamma \times \{\leftarrow; \rightarrow\}$ soit totale.

Coder des configurations Pour traduire une exécution d'une machine de Turing déterministe à l'aide de fonctions μ-récursives, il nous faut les définir sur des configurations. Celles-ci interpréteront la suite de configurations représentant cette exécution. Nous devons alors coder les configurations : on code les mots sur les rubans par la méthode de Godël et $\{\leftarrow; \rightarrow\}$ par deux entiers distincts. On peut coder les configurations de la machine de Turing à l'aide de la fonction code_conf : $\Sigma^* \times \mathbb{N} \times \Sigma^* \to \mathbb{N}^3$ définie par $(\alpha, n, \gamma) \mapsto (gd(\alpha), n, gd(\gamma))$. Comme gd est une fonction primitive récursive (elle est une somme finie de multiplications et de puissances), code_conf est également primitive récursive.

Définitions des fonctions primitives récursives utilisées lors du fonctionnement d'une machine de Turing déterministe Le comportement d'une machine de Turing déterministe peut être modélisé par une composition de fonctions primitives récursives sous une minimisation non bornée. Ces fonctions opèrent sur le codage des configurations à l'aide de la fonction code_conf qui code les configurations de $\mathcal{M}$.

INIT(x) donne la configuration initiale de $\mathcal{M}$ pour le mot d'entrée w mis sous forme de Gödel v.

$$\begin{array}{rccl} \text{INIT} : & \mathbb{N} & \to & \mathbb{N}^3 \\ & v & \mapsto & \text{code_conf}(\epsilon, 0, gd^{-1}(v)) \end{array}$$

La fonction INIT est bien primitive récursive car $\text{INIT} = \circ(\text{code_conf}, \epsilon, \mathbb{O}, \circ(gd^{-1}, \pi_1^1))$.

CONFIG_SUIVANTE(x) donne la configuration suivante d'une configuration $c = (\alpha a, c_2, b\beta)$ sous forme de Gödel que l'on note $x = (x_1, x_2, x_3)$ par $\mathcal{M}$. On définit la fonction CONFIG_SUIVANTE : $\mathbb{N}^3 \to \mathbb{N}^3$ par :

$$\text{CONFIG_SUIVANTE}(x_1, x_2, x_3) \mapsto \begin{cases} \text{code_conf}(\alpha a b', q', \gamma) & si\ \delta(c_2, b) = (q', b', \rightarrow) \\ \text{code_conf}(\alpha, q', a b' \gamma) & si\ \delta(c_2, b) = (q', b', \leftarrow) \end{cases}$$

La fonction CONFIG_SUIVANTE est une fonction primitive récursive. La fonction de transition, δ est primitive récursive car elle est à support fini. En effet, une fonction à support

fini peut toujours être écrite comme somme finie de fonctions indicatrices. Ces dernières étant, elles-même, primitives récursives par le prédicat ÉGALITÉ. On en conclut, que les fonctions à support fini peuvent être réécrites à l'aide d'une conditionnelle et de fonctions primitives récursives.

Soient (x_1, x_2, x_3) la configuration encodée de la configuration $c = (c_1, c_2, c_3)$. On note $\delta(c_2, b) = (q', b', \delta_3)$ où $\delta_3 \in \{\leftarrow; \rightarrow\}$. On récupère ces valeurs à l'aide d'une projection sur le résultat de cette fonction (ceci est primitif récursif). On a alors

$$\text{CONFIG_SUIVANTE}(x_1, x_2, x_3) = \begin{cases} (\circ(\mathsf{code_conf}, \alpha a b', q', \gamma)) & si\ \delta_3 = \rightarrow \\ (\circ(\mathsf{code_conf}, \alpha, q', a b' \gamma)) & si\ \delta_3 = \leftarrow \end{cases}$$

qui est alors primitive récursive.

CONFIG(x, n) donne la configuration après n étapes de calcul d'une configuration c sous forme de Gödel que l'on note x dans $\mathcal{M}$. On définit CONFIG : $\mathbb{N}^3 \times \mathbb{N} \rightarrow \mathbb{N}^3$ tel que :

$$\text{CONFIG}(x, n) \mapsto \begin{cases} x & si\ n = 0 \\ \text{CONFIG_SUIVANTE}(\text{CONFIG}(x, n-1)) & sinon \end{cases}$$

qui est primitive récursive car $\text{CONFIG} = rec(\pi_1^3, \circ(\text{CONFIG_SUIVANTE}, \pi_3^3))$;

STOP(x) est un prédicat qui teste si la configuration donnée $c = (c_1, c_2, c_3)$ dont la représentation de Gödel est $x = (x_1, x_2, x_3)$ est finale.

$$\begin{array}{llll} \text{STOP :} & \mathbb{N}^3 & \rightarrow & \{0; 1\} \\ & (x_1, x_2, x_3) & \mapsto & x_2 = m - 1 \end{array}$$

La fonction STOP est bien récursive primitive car $\text{STOP} = \circ(egal, \pi_2^3, m - 1)$.

SORTIE(x) donne la valeur du ruban de $\mathcal{M}$ lorsqu'elle est dans une configuration finale $c_{\text{finale}} = (c_1, c_2, c_3)$ dont la représentation de Gödel est $x = (x_1, x_2, x_3)$.

$$\begin{array}{llll} \text{SORTIE :} & \mathbb{N}^3 & \rightarrow & \mathbb{N} \\ & (x_1, x_2, x_3) & \mapsto & x_1 \end{array}$$

La fonction SORTIE est bien récursive primitive car $\text{SORTIE} = \pi_1^3$.

Représentation de $\mathcal{M}$ et conclusion La fonction f recherchée est alors $f(x) = \text{SORTIE}(\text{CONFIG}(\text{INIT}(x), \text{NB_PAS}(x)))$ où $\text{NB_PAS}(x) = \mu i.\text{STOP}(\text{CONFIG}(\text{INIT}(x), i))$. Comme NB_PAS est une minimisation non bornée d'une fonction primitive récursive $f(x, i) = \circ(\text{STOP}, \circ(\text{CONFIG}, \circ(\text{INIT}, \pi_1^2), \pi_2^2))$, elle est $\mu - recursive$. Donc f est μ-récursive par composition de fonctions primitives récursives et μ-récursive. □

Réciproquement, une fonction μ-récursive peut être simulée par une machine de Turing. Pour chacun des éléments syntaxiques nous allons construire une machine de Turing capable d'interpréter cette fonction.

Proposition 17. *Toute fonction μ-récursive est aussi calculable par une machine de Turing.*

Idée de la preuve. La preuve se fait par induction sur la structure des expressions primitives récursives.

- **Étude de la fonction nulle** Machine de Turing qui remplace les arguments par 0.
- **Étude de la fonction successeur** Machine de Turing qui remplace en incrémentant d'un le dernier nombre écrit.
- **Étude de la fonction projection** Machine de Turing qui recopie le $i^{\text{ème}}$ argument en première place et efface le reste du ruban.

— **Étude de la fonction composition** Machine de Turing qui remplace sur la deuxième machine de Turing ses arguments par le résultat de ceux-ci après exécution de la première lorsqu'ils en sont l'entrée.
— **Étude de la récursivité** Machine de Turing utilisant une boucle FOR pour exécuter le nombre de fois nécessaire la récursion.
— **Étude de la minimisation non bornée** Machine de Turing utilisant une boucle WHILE pour exécuter le nombre de fois nécessaire la minimisation.

□

Cas du λ-calcul Le λ-calcul est le deuxième modèle historique de calcul modélisant les fonctions récursives [3]. Nous montrons son équivalence aux machines de Turing via les fonctions μ-récursives. On commence par définir le langage du λ-calcul.

Définition 18 (Termes)**.** L'ensemble L des termes du λ-calcul est le plus petit ensemble tel que :

— les variables $x, y, z, \ldots$ sont des termes ;
— si u et v sont des termes, (uv) est un terme ;
— si x est une variable et t un terme, $\lambda x\, t$ est un terme.

Le terme (uv) représente l'application de la fonction u à l'argument v. Le terme $\lambda x\, t$ représente la fonction qui, à la variable x, associe le terme t : ce terme est obtenu par abstraction sur la variable x.

Exemple 19 (Codage des entiers). Le λ-calcul nous permet de coder des entiers, comme les fonctions μ-récursives avec la fonction récursive et la constante nulle. On a alors :

— $0 = \lambda(fx)x \quad -1 = \lambda(fx)fx \quad -2 = \lambda(fx)f(fx)$
— $n = \lambda(fx)\underbrace{f\,(f\,(\ldots(fx)\ldots))}_{f \text{ itéré } n \text{ fois}} = \lambda(fx)f^n x$

L'équivalence entre le modèle du λ-calcul et celui des machines de Turing se montre via les fonctions μ-récursive. On montre que le modèle du λ-calcul est équivalent à celui des fonctions μ-récursives (théorème 20). Cette équivalence est la plus simple à montrer car ces deux modèles ont été construits avec le même objectif : formaliser la récursivité. Ensuite, nous pouvons conclure de l'équivalence aux machines de Turing via l'équivalence de celles-ci aux fonctions μ-récursives (on applique le théorème 15).

Théorème 20. *Les fonctions μ-récursives sont exactement représentable par les termes du λ-calcul.*

Corollaire 21. *Les fonctions calculable par une machine de Turing déterministe sont exactement représentable par les termes du λ-calcul.*

3.3 La théorie de la calculabilité

La théorie calculabilité se fonde sur la thèse de Church, son but est de classifier les fonctions en deux catégories : les fonctions calculables et les autres. Lorsque nous étudions cette théorie du point de vue des machines de Turing, nous nous ramenons en un problème de classification sur les langages. On parle alors de la théorie de la décidabilité : nous classons les langages tels qu'ils soient ou non décidables par une machine de Turing. Dans la suite, nous nous consacrons à des problèmes que nous souhaitons classer. Pour cela nous les voyons comme des langages acceptés ou non par des machines de Turing.

Définition 22 (Machine de Turing universelle [7])**.** Une *machine de Turing universelle* peut simuler n'importe quelle machine de Turing.

Une machine de Turing universelle peut être vue comme un interpréteur de machines de Turing. En effet, comme les interpréteurs, elle prend en paramètre une autre machine de Turing que l'on applique un mot d'entrée w. La machine universelle donne alors le résultat de la machine qu'elle simule sur son entrée w.

Définition 23 (Les classes R et RE [7])**.** Un langage est dit *récursif* s'il est décidable par une machine de Turing. On définit la classe de *langages R* contenant les langages récursifs, soit $R = \{L|L \text{ est un langage décidé par une machine de Turing}\}$.

Un langage est dit *récursivement énumérable* s'il est accepté par une machine de Turing. On définit la classe de *langages RE* contenant les langages récursivement énumérables, soit $RE = \{L|L \text{ est un langage accepté par une machine de Turing}\}$.

La thèse de Church introduit l'équivalence entre calculabilité et décidabilité par une machine de Turing. Donc les classes R et RE marquent la frontière de la calculabilité. La classe R contient les problèmes décidables et donc les langages calculables. La classe RE quant-à elle contient des langages décidables (car elle contient la classe R) mais pas que. On va donner un exemple de problème qui est dans la classe RE mais qui n'est pas décidable : le problème de l'ARRÊT.

Définition 24. Le *problème de l'*ARRÊT sur une machine de Turing déterministe.

Problème	ARRÊT
entrée :	Une machine de Turing déterministe M ; un mot w
sortie	Oui si $M(w)$ s'arrête ; non sinon

Théorème 25 (Un premier problème indécidable [7])**.** *Le problème de l'*ARRÊT *est indécidable et dans RE.*

Idée de la preuve. On donne une machine de Turing universelle $\mathcal{U}$ qui accepte l'entrée $\mathcal{M}$,w si et seulement si $\mathcal{M}$ s'arrête sur w. Ceci montre que le problème de l'arrêt est récursivement énumérable.

Pour montrer que le problème de l'ARRÊT est indécidable, on raisonne par l'absurde. Il existe alors une machine de Turing $\mathcal{A}$ telle que elle s'arrête pour tout entrée $\mathcal{M}$, w et accepte une telle entrée si et seulement si $\mathcal{M}(w)$ termine. On construit une machine PARADOXE (algorithme 1). On remarque que PARADOXE(PARADOXE) ne termine pas si et seulement si $\mathcal{A}$ accepte (PARADOXE, ⟨PARADOXE⟩) où ⟨PARADOXE⟩ est le codage de la machine de Turing PARADOXE. Soit PARADOXE(PARADOXE) ne termine pas si et seulement si PARADOXE(PARADOXE) termine. Contradiction. □

Algorithm 1 La procédure PARADOXE de la preuve de l'indécidabilité du problème de l'ARRÊT.

```
1: procedure PARADOXE(M) X
2:     if A accepte (M, ⟨M⟩) then
3:         Boucler
4:     else
5:         Accepter
6:     end if
7: end procedure
```

Algorithm 2 La procédure $\mathcal{N}_{\mathcal{M},w}$ de la preuve du théorème de Rice.

```
1: procedure N_{M,w}(x)
2:     M(w).
3:     if G accepte x then
4:         accepter
5:     else
6:         rejeter
7:     end if
8: end procedure
```

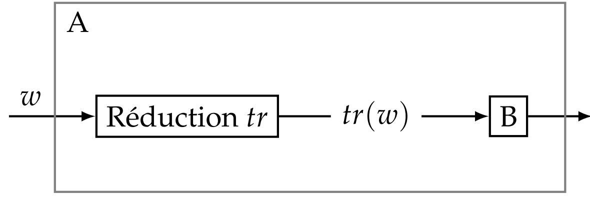

FIGURE 7 – Schéma du principe de réduction dans le cas général.

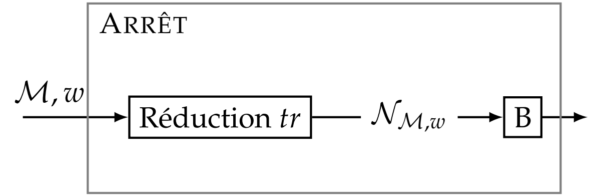

FIGURE 8 – Schéma du principe de réduction au problème de l'ARRÊT dans le cadre du théorème de Rice.

Définition 26 (Réduction d'un problème[7])**.** Une *réduction d'un problème A* à un problème B est une fonction tr calculable telle que pour toute instance w de A, w est instance positive de A si et seulement si $tr(w)$ est une instance positive de B. On dit que A se réduit à B si une telle réduction existe.

Théorème 27 (Utilisation des réductions)**.** *Si A se réduit à B, alors :*

— *B décidable implique A décidable ;*

— *A indécidable implique B indécidable.*

Définition 28. On définit le *problème de* PAVAGE DE WANG pour une famille finie de tuiles permettant de paver le plan.

Problème	PAVAGE DE WANG
entrée :	Une famille finie de tuiles carrées telle qu'on attribue à chaque triangle défini par un côté du carré une couleur
sortie	Oui si le jeu de tuiles permet de paver le plan ; non sinon

Application 29. Le PAVAGE DE WANG est indécidable

Le théorème de Rice qui est un résultat d'indécidabilité est intéressant car sa preuve met en application cette notion de réduction à une classe de problèmes. De plus, il a de nombreuses applications en informatique. En effet, les programmes informatiques (ou de manière plus abstraite les algorithmes) sont un langage dans RE (on peut tous les énumérer car on peut les dénombrer via la réduction de Gödel). Le théorème de Rice implique qu'il n'existe pas d'algorithme capable de vérifier une propriété non triviale sur les programmes comme par exemple leur correction. Ce résultat d'impossibilité théorique a introduit de nombreux champs de recherches dans la vérification formelle.

Théorème 30 (Rice [7])**.** *Pour toute propriété non triviale $\mathcal{P}$ sur les langages récursivement énumérables, le problème de savoir si le langage $L(\mathcal{M})$ d'une machine de Turing $\mathcal{M}$ vérifie $\mathcal{P}$ est indécidable.*

Démonstration. Sans perte de généralité, on suppose que $\emptyset \in \mathcal{P}$. On définit le problème $P_{\mathcal{P}}$.

Problème	$P_{\mathcal{P}}$
entrée :	Une machine de Turing $\mathcal{M}$
sortie :	Oui si $L(\mathcal{M}) \in \mathcal{P}$; non sinon

Réduisons ARRÊT à $P_{\mathcal{P}}$ (voir Figure 8). Soit $G \in \mathcal{P}$. Comme $G \in RE$, il existe une machine G qui accepte G. La réduction tr est définie par $tr(\mathcal{M}, w) = \mathcal{N}_{\mathcal{M},w}$ où $\mathcal{N}_{\mathcal{M},w}$ est la machine décrite dans l'algorithme 2.

1. tr est une fonction calculable : on construit effectivement $\mathcal{N}_{\mathcal{M},w}$ à partir de $\mathcal{M}$ et w ;
2. $(\mathcal{M}, w)$ instance positive de ARRÊT si et seulement si $\mathcal{M}$ s'arrête sur w. Comme

$$L(\mathcal{N}_{\mathcal{M},w}) = \begin{cases} G & \text{si } \mathcal{M} \text{ s'arrête sur } w \\ & \text{sinon} \end{cases}$$

$(\mathcal{M}, w)$ instance positive de ARRÊT si et seulement si $L(\mathcal{N}_{\mathcal{M},w}) \in \mathcal{P}$. Donc, $(\mathcal{M}, w)$ instance positive de ARRÊT si et seulement si $tr(\mathcal{M}, w) = \mathcal{N}_{\mathcal{M},w}$ est instance positive de $\mathcal{P}$.

□

Application 31. Si on considère $\mathcal{P} = \emptyset$, le problème LANGAGEVIDE est indécidable.

Problème LANGAGEVIDE
entrée : Une machine de Turing $\mathcal{M}$
sortie : Oui si $L(\mathcal{M}) = \emptyset$; non sinon

4 Les machines de Turing classent les fonctions calculables

Un des apports conséquents des machines de Turing est leur capacité à classer les fonctions qui sont calculables selon les difficultés à calculer ces dernières. La notion de difficulté est mesurée par la quantité de ressources nécessaires à leur calcul. Si on revient à l'analogie de l'humain effectuant un calcul, plus une fonction est compliquée à calculer plus celui-ci doit utiliser de papier et effectuer des choix pour réussir à le calculer. Dans ce contexte, nous différencions les machines de Turing déterministes et non-déterministes ce qui est une première applications des différentes variantes de machines de Turing.

Nous commençons par définir la notion de langage décidé en fonction du temps ou de l'espace. Cette mesure sur la fonction représentant la ressource utilisées va nous servir d'échelle pour créer nos classes de complexité.

Définition 32 (Arbre de calcul)**.** Un *arbre de calcul* d'une machine de Turing $\mathcal{M} = (\mathcal{Q}, \Gamma, \Sigma, \delta, s, \#, F)$ depuis la configuration (ϵ, s, w) est l'arbre de racine la configuration (ϵ, s, w) tel que les fils de tout noeud contenant la configuration c sont l'ensemble des configurations suivantes de c.

Définition 33 (Langage décidé en temps/espace f)**.** Soient $f : \mathbb{N} \to \mathbb{N}$ et une machine de Turing $\mathcal{M} = (\mathcal{Q}, \Gamma, \Sigma, \delta, s, \#, F)$. On dit que $\mathcal{M}$ *décide* L *en temps* f si $\mathcal{M}$ décide L et pour tout w, la hauteur de l'arbre depuis (ϵ, s, w) est inférieure à $f(|w|)$.

On dit que $\mathcal{M}$ *décide* L *en espace* f si $\mathcal{M}$ décide L et pour tout w, au plus $f(|w|)$ cases du ruban ont été utilisées.

Remarque 34. Attention, lorsque nous évaluons la quantité d'espace utilisé, il y a une petite subtilité si l'espace consommé par la machine de Turing est défini par une fonction f telle que $f(n) < n$. Dans ce cas, on utilise une variante des machines de Turing : la machine à deux rubans (un ruban pour l'entrée en lecture seule en une seule passe et un ruban de travail dont l'espace utilisé est majoré par f).

Cette notion de langage décidé en fonction d'un temps ou d'un espace étant définie, nous pouvons définir différentes familles de complexités qui vont nous donner les classes de complexité usuelles.

Définition 35 (Famille de complexité)**.** Soit $\mathcal{M}$ une machine de Turing déterministe et $\mathcal{N}$ une machine de Turing non-déterministe.

- $TIME(f) = \{L \mid L \text{ est décidé par } \mathcal{M} \text{ en temps } O(f(n))\}$
- $NTIME(f) = \{L \mid L \text{ est décidé par } \mathcal{N} \text{ en temps } O(f(n))\}$
- $EPACE(f) = \{L \mid L \text{ est décidé par } \mathcal{M} \text{ en espace } O(f(n))\}$
- $NSPACE(f) = \{L \mid L \text{ est décidé par } \mathcal{N} \text{ en espace } O(f(n))\}$

Définition 36 (Classes de complexité classiques)**.**

- $P = \cup_k TIME\left(x \mapsto x^k\right)$
- $NP = \cup_k NTIME\left(x \mapsto x^k\right)$
- $PSPACE = \cup_k SPACE\left(x \mapsto x^k\right)$
- $NPSPACE = \cup_k NSPACE\left(x \mapsto x^k\right)$

Dans la théorie de la complexité, l'équivalence entre une machine de Turing déterministe et une machine de Turing non-déterministe est une question importante et souvent difficile. Généralement, lorsque nous donnons deux classes de complexité telle que seule la présence ou du non-déterministe les séparent, nous ne pouvons pas dire si elles sont distinctes ou non (P égal ou non NP). Cependant, pour l'étude des classes de complexité mémoire polynomiale, nous avons le résultat suivant $NPSPACE = PSPACE$. C'est une conséquence du théorème de Savitch dont la preuve repose sur un problème de la théorie des graphes. On utilise le graphe des configurations de la machine de Turing non-déterministe (qui est borné par hypothèse), et de lui appliquer le problème d'accessibilité dans un graphe orienté. Comme celui-ci est dans P (on verra plus loin que c'est même un peu mieux que cela), on va pouvoir exploiter sa complexité spatiale afin d'obtenir le résultat que l'on recherche.

Théorème 37 (Savitch [2])**.** *Soit $f : \mathbb{N} \to \mathbb{R}^+$ telle que $f(n) \geq n$ et $f(n)$ est calculable en espace $O(f)$. Alors, $NSPACE(f) \subseteq SPACE(f^2)$.*

Démonstration. Soit $\mathcal{M}$ une machine non-déterministe qui décide un langage L en espace $f(n)$ (cela implique que $L \in NSPACE(f)$). Montrons qu'il existe une machine $\mathcal{M}'$ qui simule $\mathcal{M}$ pour décider L en espace $O(f^2(n))$.

Hypothèses sur $\mathcal{M}$ Nous supposons sans perte de généralité, sur la famille de langages décidés par la machine de Turing, que $\mathcal{M}$ possède un unique état initial q_{init} et un unique état final q_{final} qui est atteint une fois que $\mathcal{M}$ a effacé son ruban et que la tête de lecture est revenue se placer à gauche. On notera c_{accept} la configuration finale (liée à l'état final). Soit w l'entrée sur $\mathcal{M}$, on note c_{init} la configuration initiael correspondant à w.

Comme on s'intéresse à la mémoire utilisée par la machine de Turing durant son exécution (pas à son temps de calcul ou à sa taille), seule la quantité de ruban que la machine utilise durant son exécution doit être invariant par les changement effectués sur la machine de Turing. Les changements que nous effectuons ne modifient pas la quantité de ruban utilisée puisque la seule opération effectuée sur le ruban par cette nouvelle machine consiste à effacer le reste de son ruban.

Définition de $\mathcal{M}'$ On définit $\mathcal{M}'$ par l'algorithme 3. Elle prend en entrée w l'entrée de $\mathcal{M}$ et elle accepte l'exécution si $w \in L(\mathcal{M})$ (elle la rejette dans le cas contraire). On remarque que cet algorithme est bien déterministe. Pour définir la procédure $\mathcal{M}$, nous avons besoin de définir le graphe de configuration d'une machine de Turing.

Définition 38 (Graphe des configurations)**.** Le graphe des configuration est un graphe dont les sommets sont les configurations de la machine de Turing et les arêtes sont définies telles que la fonction de transition permet de passer d'une configuration à une autre en une étape.

Algorithm 3 Définition de la machine de Turing déterministe $\mathcal{M}'$

1: **procedure** $\mathcal{M}'(w)$
2:     **if** $(c_{init}) \vdash^*_{\mathcal{M}} c_{accept}$ dans le graphe des configuration de $\mathcal{M}$ **then**
3:         Accepter
4:     **else**
5:         Rejeter
6:     **end if**
7: **end procedure**

Complexité de $\mathcal{M}'$ Nous allons évaluer la complexité de $\mathcal{M}'$ en étudiant la complexité pour le calcul de ce prédicat : $(c_{init}) \vdash^*_{\mathcal{M}} c_{accept}$. On remarque dans un premier temps que si on a $(c_{init}) \vdash^*_{\mathcal{M}} c_{accept}$ alors il existe un chemin de c_{init} à c_{accept} dans le graphe des configurations de $\mathcal{M}$ que l'on note $(c_{init}) \rightarrow^*_{\mathcal{M}} c_{accept}$.

Pour ce calcul de complexité, nous allons majorer le nombre de configurations accessibles à partir c_{init} et définir une fonction qui calcul le chemin de c_{init} à c_{accept} en majorant le nombre d'étapes. Pour cela, nous allons utiliser deux lemmes.

Lemme 39. *Il existe un $d \in \mathbb{N}$ tel que le nombre de configurations accessibles depuis $c_{initial}$ soit majoré par $2^{df(|w|)}$.*

Démonstration. Le nombre de configurations accessibles depuis c_{init} est majoré par $|\mathcal{Q}| \times |\Gamma|^{f(|w|)} \times f(|w|)$ car $|\mathcal{Q}|$ est le nombre d'états que l'on peut atteindre, $|\Gamma|^{f(|w|)}$ est le nombre de mots que l'on peut écrire sur le ruban sans violer la propriété sur $\mathcal{M}$ et $f(|w|)$ est le nombre de positions possibles pour la tête de lecture. □

Lemme 40. *$c_{init} \rightarrow^* c_{accept}$ si et seulement si $c_{init} \rightarrow^{2df(|w|)} c_{accept}$.*

Nous allons alors nous appuyer sur la fonction CHEMIN ? dont on donne la spécification. Pour deux configurations c_1 et c_2 de $\mathcal{M}$ et ainsi qu'un entier t, CHEMIN ?(c_1, c_2, t) renvoie VRAI s'il existe un chemin (donc une suite de configurations pour $\mathcal{M}$) de c_1 vers c_2 dans le graphe des configurations de $\mathcal{M}$ en au plus t étapes ; et FAUX sinon. Dans la suite, on suppose que t est une puissance de 2 (dans le cas contraire, comme on a toujours l'existence d'une puissance de 2 qui est plus grande que t, on prend la plus petite de ces puissances de 2 comme nouvelle valeur de t).

Démonstration. Le sens indirect est immédiat. Nous allons alors montrer la réciproque.

Nous implémentons le test $c_{init} \rightarrow^* c_{final}$ à l'aide de la fonction CHEMIN ?$(c_{initial}, c_{accept}, 2^{df(|w|)})$ où CHEMIN ? est définie comme précédemment et par l'algorithme 4.

La fonction CHEMIN ? suit le paradigme diviser pour régner. Sa complexité spatiale est alors de $Cout(t) = O(f) + Cout(\frac{t}{2})$ où t est le troisième argument de CHEMIN ? (Hypothèse 2 : on peut calculer $f(|w|)$ en $O(f)$). On obtient donc $Cout(t) = O(f \log_2 t)$ (par le master théorème sur la complexité du principe diviser pour régner). La complexité spatiale de $\mathcal{M}'$, par l'hypothèse, est de $O(f) + Cout(2^{df(|w|)}) = O(f) + O(f \log_2(2^{df(|w|)}) = O(f) + O(fdf) = O(f) + O(f^2) = O(f^2)$ car on calcul une fois $f(|w|)$ qui se calcule en $O(f)$, par hypothèse, pour la fonction CHEMIN ?. On exécute alors la fonction CHEMIN ? avec $t = 2^{df(|w|)}$ ce qui donne la complexité annoncée. □

□

Corollaire 41. *$NPSPAPCE = PSPACE$*

Algorithm 4 Fonction CHEMIN ?

```
1: function CHEMIN ?(c_1, c_2, t)
2:     if t = 1 then
3:         Renvoie c_1 →^{≤1} c_2
4:     else
5:         for toute configuration c de taille de ruban d'au plus f(|w|) do
6:             if CHEMIN ?(c_1, c, t/2) et CHEMIN ?(c, c_2, t/2) then
7:                 Renvoie VRAI
8:             end if
9:         end for
10:        Renvoie FAUX
11:    end if
12: end function
```

Une version plus générale du théorème de Savitch existe, nous l'énonçons ci-dessous. La preuve de celui-ci se fait de manière analogue. La seule différence vient au moment de la définition de la fonction CHEMIN ? où nous devons être plus pointilleux sur sa définition afin de respecter les complexités annoncées. Pour contrer cela, on appelle la fonction CHEMIN ? pour chaque valeur de $f(i)$ et on incrémente i pas à pas tant qu'on n'a pas trouvé de chemin acceptant. Par ce procédé, nous calculons expérimentalement la borne que l'on a ajoutée dans les hypothèses. De plus, l'hypothèse $f(n) \geq n$ nous permet de lire l'entrée sur le ruban ; cependant avec une machine à plusieurs rubans, $f(n) \geq \log n$ suffit.

Théorème 42 (Théorème de Savitch [4])**.** *Soit $f : \mathbb{N} \rightarrow \mathbb{R}^+$ tel que pour n assez grand $f(n) \geq \log n$. Toute machine de Turing non-déterministe qui décide en espace $f(n)$ est équivalente à une machine de Turing déterministe en espace $O(f^2(n))$.*

Définition 43. On définit le problème UNIVERSALITÉ sur les langages rationnels.

Problème	UNIVERSALITÉ
entrée :	Une expression rationnelle E
sortie	Oui si $L(E) = \Sigma^*$; non sinon

Application 44. Le problème UNIVERSALITÉ est un problème dans *PSPACE*

Pour les autres classes de complexités, l'équivalence entre les machines de Turing déterministes et les machines de Turing non-déterministes est une question ouverte. Dans le cas des langages décidables en temps polynomiales avec une machine de Turing déterministe ou non, la question est restée ouverte. Nous allons alors définir la notion de la *NP*-complétude qui caractérise les problèmes les plus difficiles de la classe *NP* et qui ne sont à priori pas dans *P*.

Définition 45 (*NP*-complétude [2])**.** Un langage L est NL-dur si et seulement si tout problème dans NP se réduit en temps polynomial à L. Si, de plus, L est dans NP, L est dite NP-complet.

Définition 46. On définit le problème SAT sur les formules de la logique propositionnelle.

Problème	SAT
entrée :	Une formule ϕ de la logique propositionnelle
sortie	Oui si ϕ est satisfiable ; non sinon

Théorème 47 (Cook [2])**.** *Le problème* SAT *est NP-complet.*

Il arrive que la machine de Turing non-déterministe ne soit pas la seule variante que l'on doit utiliser pour définir des classes de complexité : les machines de Turing déterministes ou

non avec plusieurs rubans permettent de définir de nouvelles classes de complexité. Pour définir une machine de Turing qui décide en espace logarithmique, il nous faut une machine de Turing à deux rubans : un ruban qui contient l'entrée en lecture seule en une seule passe et le second de travail qui contient la borne logarithmique.

Définition 48 (Classes NL et L [2])**.**

— $L = SPACE(\log n)$

— $NL = NSPACE(\log n)$

Définition 49. On définit le problème ACCESSIBILITÉ sur les graphes orientés.

Problème	ACCESSIBILITÉ
entrée :	Un graphe orienté G est deux sommets s et t
sortie	Oui si il existe un chemin de s à t dans G ; non sinon

Proposition 50. *Le problème* ACCESSIBILITÉ *est dans* NL.

Idée de la preuve. On a une machine (algorithme 5) en espace logarithmique qui décide en espace logarithmique le problème de l'ACCESSIBILITÉ. □

Algorithm 5 La procédure PATH ? de la preuve de l'appartenance du problème de l'ACCESSIBILITÉ à la classe NL.

```
1: procedure PATH ?(()G, s, t)
2:     while dos ≠ t
3:         s ← un successeur de s
4:     end while
5: end procedure
```

Remarque 51. Si G est un graphe non-orienté alors le problème arrive dans L.

5 Conclusion

Les machines de Turing sont un modèle de calcul riche possédant de nombreuses applications. Lors de leur introduction, les machines de Turing apportent un nouveau point de vue simple et original qui a plusieurs conséquences. On obtient un modèle mécanique (en opposition aux modèles précédents plus mathématiques) qui est aujourd'hui considéré comme l'abstraction de nos ordinateurs. De plus, elles travaillent sur des langages (qui s'abstrait de $\mathbb{N}$). Ceux-ci peuvent être acceptés ou décidés ce qui est la notion centrale des applications aux machines de Turing.

Les machines de Turing font partie de la famille de modèle de calcul qui sont les plus expressifs : c'est un modèle très riche. Il existe d'autres modèles dans cette famille : les fonctions μ-récursives ou le λ-calcul comme le montre leur équivalence aux machines de Turing déterministe au sens de l'expressivité. Comme, pour ces différents modèles, elles permettent de définir la frontière conceptuelle de la calculabilité : de ce qu'on est capable de calculer ou non. Les machines de Turing n'ont pas révolutionnée la théorie de la calculabilité. Elles l'ont, cependant consolidé en donnant un autre point de vu à cette théorie. On est alors passé des fonctions généralement définies sur des entiers à des langages définis sur un alphabet fini quelconque. Les extensions d'une machine de Turing ne permettent pas de décidé de nouveau langages. Cette robustesse et l'équivalence avec les autres modèles de calcul à la base de la théorie de la calculabilité conforte cette théorie qui nous permet aujourd'hui de classifier les problèmes

selon s'il admettent ou non un procédé permettant de les résoudre avec un ordinateur : s'ils admettent un algorithme.

Les machines de Turing sont considérées comme le modèle abstrait des ordinateurs. Une question naturelle lorsqu'on a cet outil entre les mains est *quelles sont les ressources nécessaires pour répondre à un problème ?*. Cette question en amène une autre *les extensions des machines de Turing sont-elles équivalentes lorsqu'on considère la quantité de ressources nécessaire pour décider un langages ou permettent-elles de classer les problèmes ?*. Ces questions sont à l'origine de la théorie de la complexité qui donne une échelle de difficulté selon la variante de la machine de Turing utilisées pour les problèmes. La théorie de la complexité possède encore de nombreuses questions ouvertes comme notamment la célèbre question P est-il ou non égal à NP. Dans cette théorie, les extensions de machines de Turing sont supposées non équivalentes et utiliser une machine de Turing déterministe ou non donne une indication sur les difficultés du problème.

Les théories de la calculabilité et de la complexité sont des théories riches en modèles : on définit des classes de complexité à l'aide de circuits booléens ou de la logique. Ceux-ci ont été introduits afin de répondre aux questions de la théorie de la complexité. La plupart de ces nouveaux modèles sont également utilisés dans la théorie de la calculabilité. Ces modèles caractérisent la frontière de la calculabilité (comme la logique du premier ordre ou les circuits booléens). De plus, nous n'avons pas présenté toutes les variantes des machines de Turing qui affinent les classes de complexité. On peut citer par exemple les machines de Turing à oracle ou encore les machines de Turing alternantes qui viennent enrichir la théorie de la complexité ou encore donnent de nouvelles variantes équivalentes à une machine de Turing déterministe.

Références

[1] J.-Y. Girard A. Turing. *La machine de Turing*. Source du savoir, Seuil, 1991.

[2] O. Carton. *Langages formels, calculabilité et complexité*. Vuibert, 2008.

[3] M. de Rougemont R. Lassaigne. *Logique et fondements de l'informatique. Logique du* 1er *ordre, calculabilité et* λ*-calcul*. Hermes, 1993.

[4] M. Sipser. *Introduction to the Theory of Computation*. Cengage learning, 1133187811.

[5] J. Stern. *Fondements mathématiques de l'informatique*. Ediscience internationnal, 1990.

[6] Alan Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(1) :230–265, 1936.

[7] P. Wolper. *Introduction à la calculabilité*. Dunod, 2006.

A Gödelisation

Pour représenter des chaînes de caractères par des entiers ou vise versa [7], on peut raisonner comme suit : les chaînes de caractères est un ensemble dénombrable donc il existe une bijection entre les entiers et les chaînes de caractères. On peut alors coder les entiers (ou les chaînes de caractères) à l'aide de cette bijection (ou de sa réciproque). Ce raisonnement est correct mais pas suffisant : dans le contexte de la calculabilité, il faut de plus que cette bijection (et sa réciproque) soit calculable. Nous nommerons cette représentation, la représentation effective des entiers ou des chaînes de caractères.

Lemme 52. *Il existe une représentation effective des chaînes de caractères par les entiers.*

Démonstration. Les représentations binaire ou décimale sont des exemples de représentations effectives. Nous allons montrer que la représentation binaire est bien une représentation effective en montrant qu'il existe des fonctions μ-récursives (en montrant qu'elles sont des fonctions primitives récursives) qui permettent de calculer la représentation binaire.

BINLONG(n) Calcul la longueur binaire de n. C'est une fonction primitive récursive car c'est une minimisation bornée (par n) de division par deux qui donne le résultat.

CHIFFRE(n, m) Donne la $m^{\text{ième}}$ chiffre de l'entier n si $m \leq \text{BINLONG}(n)$ et 0 sinon. C'est une fonction primitive récursive car c'est une minimisation bornée (par m) de division par deux qui donne le résultat.

Ces deux fonctions sont bien primitives récursives. □

Remarque 53. Notons que cette représentation bien que calculable par une machine de Turing n'est pas nécessairement la plus agréable à manipuler. Une représentation unaire, vue comme une composition de fonction successeur, peut être beaucoup plus intéressante pour écrire sur le ruban de notre machine de Turing.

La représentation inverse est la plus délicate, c'est elle qui porte le nom de Gödélisation car introduite par le logicien Kurt Gödel.

Lemme 54. *Il existe une représentation effective des entiers par les chaînes de caractères.*

Démonstration. Soit Σ un alphabet contenant k symboles. Une représentation simple consiste à attribué à chacune des lettres de l'alphabet un nombre entre 0 et $k-1$ (si $a \in \Sigma$, on note $gd(a)$ son codage par un entier). Puis on code la chaîne de caractère à partir de ce codage : soit $w = w_1 \dots w_n$ une chaîne de caractères, alors $gd(w) = \sum_{i=0}^{l} k^{l-i} gd(w_i)$ qui correspond au naturel dont la représentation en base k est $gd(w_1) \dots gd(w_n)$.

Cependant, ce codage n'est pas univoque puisque $gd(aa) = 00 = 0 = gd(a)$. Pour corriger cette représentation, il faut que le codage ne'utilise pas 0. Pour cela on code les lettres de l'alphabet entre 1 et k et on a $gd(w) = \sum_{i=0}^{l} (k+1)^{l-i} gd(w_i)$ □

Remarque 55. Il est bon de remarquer que la représentation donnée par gd n'est pas injective. En effet, par définition 0 ne sera jamais atteint. Cependant, comme nous cherchons une représentation des chaînes de caractères et non des entiers, cette non injectivité n'est pas gênante.