Chapitre I Introduction

En informatique, on s'équipe d'une *outil de calcul* (l'ordinateur) pour effectuer un certain travail. Il est légitime de se poser ce qui sera notre "question fondamentale" :

qu'est-il possible de faire avec cet outil ?

Au cours de IFT 311, on a pu répondre à cette question pour certains outils qui servaient de modèle théorique simplifié pour les algorithmes et les ordinateurs. Ainsi, en posant la question ci-dessus au sujet des automates finis on obtenait la réponse :

on peut reconnaître les langages réguliers, et rien d'autre.

La même question, posée pour les automates à pile, donnait :

on peut reconnaître les langages hors-contexte, et rien d'autre.

En fin de trimestre, on a posé la même question au sujet des machines de Turing ; les choses sont devenues plus complexes, on a dû établir une distinction entre *langages décidés* et *langages acceptés* et la preuve que les deux notions ne sont pas équivalentes n'utilisait pas un lemme de l'étoile, comme dans les cas précédents, mais un argument plus subtil. Le résultat peut se formuler ainsi :

il existe des langages qui ne sont acceptés par aucune machine de Turing ;
il en existe qui ne sont décidés par aucune machine de Turing.

On reviendra sur cette question. On verra aussi que notre "question fondamentale" peut être posée pour des systèmes qui ne sont pas des modèles de calcul mécanique mais plutôt des modèles de calcul abstrait ou de description d'objets abstraits : des théories de la logique, des théories de l'arithmétique des nombres entiers et des théories pour le calcul de fonctions. Dans tous les cas, poser cette question mènera à une réponse semblable à celles citées ci-dessus : ces théories ne sont pas toutes-puissantes et il y a des choses qu'elles ne peuvent pas faire. De plus, on pourra se convaincre que certaines des réponses obtenues sont équivalentes, en ce qu'elles expriment la même réalité dans des formalismes différents.

Un ordinateur est une machine qui exécute mécaniquement des *algorithmes* encodés sous la forme de programmes. Il n'existe pas de définition formelle (mathématique) de la notion d'algorithme, mais on peut en donner des approximations, comme celle-ci.

Définition 1.1 Un *algorithme* est une suite finie d'instructions bien définies qui permet de résoudre en problème donné ou d'effectuer un travail bien défini sur les données qu'on lui soumet.

Exemple 1.2 Tâches pouvant être effectuées par un algorithme.

- trier des nombres ou des chaînes de caractères ;
- déterminer si une suite de bits est un palindrome ;
- calculer une transformée de Fourier discrète ;

- compiler un programme écrit en `C++`.

Certains algorithmes retournent une réponse binaire OUI/NON : il *décident* l'appartenance de la donnée à un *langage*. En général, le résultat est plus gros qu'un simple bit : il peut être vu comme le résultat d'une fonction appliquée à la donnée et calculée par l'algorithme.

Un algorithme calcule une fonction.

On est alors amené à se demander :

quelles sont les fonctions qui peuvent être calculées par un algorithme ?

et parler de "fonction calculable," d'où les questions :

existe-t-il une définition formelle pour la notion de fonction calculable ?
Et s'il y a plusieurs définitions, sont-elles équivalentes ?

Quand un algorithme décide si une donnée $w \in \Sigma^*$ appartient à un langage $L \subset \Sigma^*$, il calcule en fait la fonction caractéristique de L : une fonction dont le domaine est l'ensemble Σ^* et dont le codomaine est la paire $\{V, F\}$ (ou $\{0, 1\}$, selon la convention choisie). La notion de fonction à valeur dans $\{V, F\}$ permet d'établir un lien avec la logique.

Définition 1.3 Un *prédicat* est une proposition contenant des variables et susceptible de prendre la valeur V ou F ; on peut aussi le voir comme étant une fonction à valeur dans $\{V, F\}$.

On peut se poser les questions :

quels sont les prédicats qui peuvent être calculés par un algorithme ?
comment définir formellement ce qu'est un prédicat calculable ?

Nos questions établissent un lien informel entre le calcul mécanique (machines de Turing et algorithmes), la logique formelle et les fonctions entières. Au cours des prochains chapitres on va

- préciser et définir formellement les notions de
  - *calcul mécanique*, par le biais des *machines de Turing* et des notions connexes de *langages décidables, langages récursivement énumérables* et de *fonctions calculables* au sens de Turing ;
  - *fonction entière* pouvant être décrite de façon finie, entre autres les *fonctions récursives primitives* et *récursives générales* ;
  - *théorie mathématique*, entre autre le *calcul propositionnel*, la *logique du premier ordre* et l'*arithmétique de Peano* ;
- établir des liens informels et des équivalences formelles entre ces notions ;
- démontrer qu'il existe une *limite* au pouvoir descriptif des classes d'objets ainsi définies : il y a des fonctions qui ne sont pas récursives générales, il existe des formules de l'arithmétique de nombres entiers qui ne peuvent être démontrées dans le cadre de la théorie de l'arithmétique de Peano (*théorèmes de Gödel*) et il y a des fonctions qui ne sont pas calculables mécaniquement, entre autre la fonction caractéristique de certains langages (*théorème de Turing*).

Remarques bibliographiques

On indique ici les sources où la matière a été prise ainsi que celles où il est possible d'aller chercher un complément d'information. *Il importe de savoir que les présentes notes sont conçues pour être utilisées en conjugaison avec un manuel de référence (au choix de l'étudiant) pour les chapitres II à IV, et indépendamment de tout livre pour les chapitres V à VII.*

Machines de Turing et indécidabilité

La matière du chapitre II est couverte de manière détaillée dans les deux manuels couramment utilisés pour le cours IFT311 ([16, chapitre 9] et [10, chapitres 9 et 10]). Il en est de même pour le chapitre III, sauf pour la section sur la relativisation qui est adaptée de [14, pages 210-212].
Pour des lectures complémentaires : la matière des deux chapitres concernés est bien traitée dans [18], quoique de manière un peu concise ; on trouvera dans [9] un contenu complet et un bon choix d'exercices ; le manuel [6] offre des explications présentées de façon intéressante ; enfin, [15] offre un traitement différent, de plus haut niveau. La référence la plus citée dans le domaine est [14], un livre très dense qui couvre beaucoup plus de matière que les autres.

Fonctions récursives

Les meilleures références, de loin, sont [16, chapitre 13], [12, chapitre 3] et [8] (à distinguer de [9], qui ne traite pas ce sujet). D'autres livres comme [10, chapitre 13] et [18, chapitre 6] abordent aussi ce sujet, mais de façon plus superficielle.

Théorie de la complexité

En ce qui concerne la **NP**-complétude, la référence canonique est le livre [**?**]. Le sujet est aussi traité dans les manuels [6, 8, 9, 10, 16, 18]. Le livre [15] donne une description solide et succincte de la théorie de la complexité. Plusieurs titres abordent la théorie de différentes manières, parmi lesquels [1, 2, 5, **?**, 17] ; on en trouvera la plupart à la bibliothèque.

Références complémentaires sur la calculabilité

Incomplétude de l'arithmétique. Ce sujet est lié de près à celui de la décidabilité. Il est traité, à des niveaux techinques très différents, dans [3, 11, 12, 16].
Le lecteur qui veut en savoir plus sur la calculabilité pourra consulter les deux livres d'Odifreddi [13] ; il constatera que le cours ne présente qu'une introduction à un domaine qui a été abondamment étudié au cours du XXe siècle.

Chapitre II Les machines de Turing

On poursuit l'études des langages formels en définissant les machines de Turing, un modèle de calcul qui reproduit exactement le pouvoir des ordinateurs réels, et en leur associant deux classes de langages : décidables et récursivement énumérables. On montre ensuite que la première de ces classes est un sous-ensemble strict de la seconde, et que celle-ci ne coïncide pas avec l'ensemble de tous les langages, ce qui implique l'existence de problèmes pour lesquels il n'existe aucune solution algorithmique.

Les machines de Turing et leurs principales propriétés font l'objet d'une discussion dans presque tous les manuels de théorie des langages formels ; cependant, les notations et certaines conventions varient d'un auteur à l'autre. Aussi, le présent chapitre se bornera à fixer le vocabulaire et le formalisme qui seront utilisés par la suite.

1 Le modèle de base

Informellement, une machine de Turing "standard" est constituée d'une unité centrale de contrôle pouvant entrer dans un état pris parmi un nombre fini de possibilités, d'un ruban semi-infini dont les cases contiennent chacune un symbole pris dans un alphabet fini Γ, ainsi que d'une unique tête de lecture-écriture pouvant se déplacer dans les deux directions.

Définition 2.1 Une *machine de Turing* est un quintuplet $(Q, \Gamma, \Sigma, \delta, q_0)$ où

Q est l'ensemble des états du contrôle central ;

Γ est l'alphabet des symboles de ruban ;

$\Sigma \subset \Gamma$ est l'alphabet dans lequel est écrite la donnée reçue en entrée ;

$\delta \subset (Q \times \Gamma) \times (Q \times \Gamma \times \{G, D\})$ est une fonction partielle (*fonction de transition*) ; ici, les lettres G et D dénotent un déplacement de la tête vers la gauche et la droite, respectivement ;

$q_0 \in Q$ est l'état initial.

L'ensemble $(\Gamma \setminus \Sigma)$ contient un caractère $\square$ placé au départ du calcul dans toutes les cases du ruban qui ne contiennent aucune donnée (*symbole vide*).

On remarque que la machine est déterministe, en ce qu'il existe au plus une transition possible pour chaque couple de $Q \times \Gamma$; lorsque pour un couple donné (q, γ) la valeur de $\delta(q, \gamma)$ n'est pas définie, alors par convention le calcul de la machine s'arrête.
Les machines de Turing obéiront la plupart du temps aux conventions suivantes :

- il y aura un nombre fixe de rubans semi-infinis, chacun portant des cases numérotées en ordre croissant, habituellement à partir de 0 ;
- sur chaque ruban, la case 0 portera un symbole-butoir spécial afin d'éviter les déplacements à gauche de la case 0 ;
- il y aura exactement une tête de lecture-écriture par ruban ; toutes les têtes de la machine liront, écriront et se déplaceront de manière synchrone ;
- s'il existe dans Q des états à partir desquels aucune transition n'est possible, on les nommera *états d'arrêt*, et le cas échéant on partagera ceux-ci entre *états accepteurs* et *états non-accepteurs* ;
- enfin, on pourra ajouter à G et D un troisième choix S pour les transitions sans déplacement.

Comme pour le cas des automates finis et des automates à pile, il existe une représentation visuelle sous la forme d'un graphe orienté dont chaque sommet représente un des états et chaque arête une des transitions (*diagramme des transitions*) ; l'étiquette d'une arête de p vers q sera a/bX lorsque l'arête représentera le fait que $\delta(p, a) = \langle q, b, X\rangle$, avec $a, b \in \Gamma$ et $X \in \{G, D\}$.
On définit maintenant les notions complémentaires qui permettent d'étudier la capacité de calcul du modèle.

Définition 2.2 Pour une machine de Turing définie selon nos conventions, une *configuration* est la description du contenu non-trivial des rubans, de l'état du contrôle central et de la position des têtes de lecture-écriture. Pour un ruban donné dont les cases sont numérotées à partir de 0, on représentera l'information pertinente sous la forme uqv, où

$u \in \Gamma^*$ est le contenu du ruban allant de la case 1 jusqu'à la case immédiatement à gauche de la tête ;

$q \in Q$ est l'état dans lequel se trouve le contrôle central ;

$v \in \Gamma^*$ est le contenu du ruban à partir de la case sur laquelle pointe la tête et en allant vers la droite jusqu'au dernier symbole non-trivial.

On distingue entre autres cas particuliers la *configuration initiale* sur la donnée w, à savoir q_0w : la tête est sur la case 1 et la donnée w est enregistrée sur les cases 1 à $|w|$; une *configuration d'arrêt* (resp. *acceptante*, *de rejet*) en est une où le contrôle central se trouve dans un état d'arrêt (resp. accepteur, non-accepteur).

Exemple 2.3 (*i*) Une machine à un ruban qui efface sa donnée et arrête dans un état h avec la tête sur la case 1 commence son calcul à la configuration q_0w et le termine dans $h\square$. (*ii*) Soit une machine à un ruban qui dédouble sa donnée en séparant les deux copies par un espace vide et qui termine son calcul dans un état h avec la tête sur la première case à droite de la copie : une telle machine fait un calcul qui commence à la configuration q_0w et qui termine dans $w\square wh\square$.

(*iii*) Pour une machine à deux rubans qui commence avec la donnée sur le premier ruban et rien sur le second, la configuration initiale est un couple de la forme $(q_0 w,\ q_0\Box)$.

Définition 2.4 Relation de transition entre deux configurations d'une machine de Turing à un ruban. Avec $C = uqv$ et $C' = u'q'v'$ on a $C \vdash C'$ ("transition en une étape") ssi on a un des cas suivants :

1. (déplacement à gauche) on a $a, b, c \in \Gamma$ et $w \in \Gamma^*$ tels que $u = u'c$, $v = aw$, $v' = cbw$ et $\delta(q, a) = \langle q', b, G\rangle$;
2. (déplacement à droite) on a $a, b \in \Gamma$ tels que $u' = ub$, $v = av'$ et $\delta(q, a) = \langle q', b, D\rangle$.

La fermeture réflexive et transitive $\vdash^*$ de cette relation encode l'existence d'un calcul fini menant de C à C′. On définit également la relation $\vdash^+$ qui encode l'existence d'un calcul fini non-trivial menant de C à C′.

Un point essentiel à remarquer est qu'une machine de Turing peut très bien ne jamais arrêter ; on dénotera par $C \vdash^* \infty$ le fait que le calcul à partir de C est infini. Les deux manières de ne jamais arrêter sont :

- d'entrer dans une boucle infinie, c'est à dire de passer par une configuration C pour laquelle on a $C \vdash^+ C$;
- d'entrer dans un déplacement infini vers la droite, c'est à dire de passer dans une configuration uqv telle que pour tous u_1, q_1, v_1 tels que $\vdash^* u_1 q_1 v_1$, il existe u_2, q_2, v_2 tels que $|u_2| > |u_1|$ et $u_1 q_1 v_1 \vdash^+ u_2 q_2 v_2$.

Il y a donc trois conclusions possibles concernant le calcul d'une machine M sur sa donnée :

- elle arrête dans un état accepteur, ou
- elle arrête dans un état non accepteur, ou
- elle n'arrête jamais.

2 Variantes

Il existe maintes possibilités de variantes ; bien qu'elles permettent d'obtenir une accélération du calcul ou une plus grande simplicité dans la description de l'algorithme sous-jacent, c'est un exercice standard de démontrer qu'elles n'augmentent pas de façon absolue la puissance de calcul des machines de Turing.

Définition 2.5 Variations sur le modèle de la machine de Turing.

- Modèle de base : un seul ruban et une seule tête de lecture-écriture.
- Machine avec un nombre fixe de rubans, chacun équipé d'une unique tête de lecture-écriture.
- Machine avec un nombre fixe de rubans, chacun équipé de k têtes de lecture-écriture, avec k une constante.

- Machine avec un unique ruban infini des deux côtés ; les cases sont numérotées avec un élément de $\mathbb{Z}$; les conventions concernant les configurations sont adaptées en conséquence.
- Machine avec un unique ruban semi-infini à deux dimensions : chaque case est numérotée avec un couple de $\mathbb{N} \times \mathbb{N}$, les mouvements possibles sont G, D, H et B, si bien que les cases accessibles en une étape à partir de (m, n) sont $(m-1, n)$, $(m+1, n)$, $(m, n-1)$ et $(m, n+1)$. Il arrive souvent que ce "ruban" bidimensionnel soit conceptuellement découpé en "lignes," la ligne n étant la suite de toutes les cases numérotées (n, i), $i \in \mathbb{N}$; ceci est particulièrement utile lorsque la machine simule en parallèle un nombre variable de calculs d'un machine à un ruban.
- Versions non-déterministes des machines ci-dessus : le non-déterminisme consiste à permettre que pour chaque couple de $Q \times \Gamma$ il existe plus d'une transition possible ; la machine choisit l'une d'elles de la même manière que le fait un automate fini non-déterministe ; on distingue alors entre l'existence et la non-existence d'un calcul fini (ou accepteur, selon le contexte) parmi tous ceux qui sont possibles au départ d'une même configuration.

Théorème 2.6 Toutes les variantes déterministes ci-dessus peuvent être simulées exactement par une machine de Turing du modèle de base.
D'autre part, pour toute machine non-déterministe M, pour tout état q de M et toute donnée w, on peut construire une machine déterministe M' qui, à partir de la configuration initiale $q_0 w$, pourra déterminer en un temps fini si M passe par l'état q au cours d'au moins un des calculs possibles à partir de $q_0 w$. En particulier, il est possible de construire une machine M' qui arrête exactement sur les mêmes données que M.

La principale conséquence de ce théorème est de faire de la machine de Turing le modèle de calcul qui donne la meilleure abstraction des algorithmes et des ordinateurs réels, étant donné que le fonctionnement de ceux-ci peut toujours être modélisé sous une forme qui s'apparente à une machine de Turing disposant d'un espace-mémoire plus ou moins compliqué à décrire. L'avantage du modèle de base est de faciliter le travail abstrait et la démonstration d'éventuels théorèmes ; celui des variantes est de simplifier la description d'une machine pour la réalisation d'un travail donné. On résume ces considérations sous la forme suivante.

> **Thèse de Church-Turing :** les machines de Turing ont exactement la même puissance de calcul que ce qui correspond à notre notion intuitive des algorithmes et des ordinateurs réels.

3 Emploi des machines de Turing

On peut utiliser les machines de Turing de trois manières différentes :

- pour reconnaître un langage ;
- pour calculer une fonction ;

- pour énumérer un langage.

Quand il s'agit de reconnaître un langage, il existe deux conventions, qui amènent à définir deux grandes classes de langages. À une machine M donnée possédant au plus un état d'arrêt accepteur q_a et au plus un état d'arrêt non-accepteur q_r, on associe les sous-ensembles de Σ^* suivants :

- $L_A(M) = \{\, w \in \Sigma^* \mid \exists u, v \in \Gamma^* : q_0 w \vdash^* u q_a v \,\}$, le langage des mots acceptés par M ;
- $L_R(M) = \{\, w \in \Sigma^* \mid \exists u, v \in \Gamma^* : q_0 w \vdash^* u q_r v \,\}$, le langage des mots rejetés par M ;
- $L_\infty(M) = \{\, w \in \Sigma^* \mid q_0 w \vdash^* \infty \,\}$, le langage des mots sur lesquels M n'arrête jamais.

Définition 2.7 Conventions pour la reconnaissance de langages.
Un langage $K \subset \Sigma^*$ est *décidé* par la machine M ssi

i. M arrête sur toutes ses données : $L_\infty(M) = \emptyset$, et
ii. $K = L_A(M)$.

Un langage $K \subset \Sigma^*$ est *accepté* par la machine M ssi $K = L_A(M)$; dans ce cas-ci, il se peut que $L_\infty(M) \neq \emptyset$.
On dit qu'un langage est *décidable* lorsqu'il existe une machine de Turing qui le décide, et *acceptable au sens de Turing* lorsqu'il existe une machine de Turing qui l'accepte. On définit ainsi les classes `Dec` et `RE`.

Noter que lorsqu'il s'agit d'accepter un langage, on construit presque toujours M de manière à avoir $L_R(M) = \emptyset$: un mot w appartient alors au langage accepté par M si, et seulement si, le calcul de M arrête sur la donnée w. Cette restriction n'a aucune conséquence sur la définition des classes `Dec` et `RE`.
Avec les notations `Reg` et `HC` pour les langages réguliers et hors-contexte, respectivement, on sait dès maintenant que les relations suivantes sont valides :

$$\texttt{Reg} \subset \texttt{HC} \subset \texttt{Dec} \subseteq \texttt{RE} \subseteq \mathcal{P}(\Sigma^*).$$

Les questions auxquelles on répond plus loin consistent à décider si toutes ces inclusions sont strictes ; en d'autres termes, il s'agit de savoir ce qu'il est possible de faire faire par une machine de Turing en un nombre fini d'étapes de calcul.

En ce qui concerne le calcul de fonctions, on simplifie le modèle en donnant à la machine un unique état d'arrêt q_a ; on spécifie de plus que M peut entrer dans q_a uniquement de manière à ce que la tête pointe sur la case 1 ; la convention se lit alors comme ceci.

Définition 2.8 Une fonction $f : \Sigma^* \to \Sigma^*$ est *calculée* par la machine M ssi

i. quand $f(w)$ est définie, M arrête sur la donnée w avec $f(w)$ sur son ruban, c'est à dire : $q_0 w \vdash^* q_a f(w)$;
ii. quand $f(w)$ n'est pas définie, alors le calcul de M sur la donnée w est infini : $q_0 w \vdash^* \infty$.

On dit qu'une fonction est *calculable au sens de Turing* lorsqu'il existe une machine de Turing qui la calcule.

On étudiera ces fonctions plus en détail dans le chapitre sur les fonctions récursives.

Il existe une troisième manière d'utiliser une machine de Turing : comme outil pour donner la liste complète des mots d'un langage ; on dit que la machine énumère ce langage.

Définition 2.9 Un langage $K \subset \Sigma^*$ est *énuméré* par la machine M ssi on a identifié dans M un état q_e tel que $(\forall\, w \in \Sigma^*)[\, w \in K \Leftrightarrow (\exists\, v \in \Gamma^*)[q_0\square \vdash^* wq_ev]\,]$. Autrement dit, la machine M part avec un ruban vide et, à chacun de ses passages dans l'état q_e au cours d'un calcul possiblement infini, nous donne sur son ruban à gauche de la tête de lecture-écriture un mot de K. On dit qu'un langage est *récursivement énumérable* lorsqu'il existe une machine de Turing qui l'énumère.

Ce mode d'utilisation nous fournit exactement la même puissance descriptive que la convention d'acceptation définie plus haut.

Théorème 2.10 Un langage est acceptable au sens de Turing ssi il est récursivement énumérable.

Démonstration. Soit K un langage énuméré par une machine M. On construit une machine M' équipée de deux rubans : le premier contient la donnée tandis M est simulée sur le second ruban ; à chaque fois qu'un nouveau mot est énuméré, M' passe dans une procédure qui compare ce mot avec le contenu du premier ruban, et arrête si les deux sont identiques.
Soit $K \subset \Sigma^*$ un langage accepté par une machine N. On construit une machine N' équipée d'un "ruban" bidimensionnel dont les lignes sont numérotées à partir de -1. On utilise aussi une numérotation de Σ^*, c'est à dire une bijection de Σ^* vers $\mathbb{N}$. Au départ le ruban est entièrement vide ; par la technique du "dovetailing," la machine N' simule N sur tout Σ^* en traitant sur chaque mot w de Σ^* sur la ligne de ruban dont le numéro est le même que celui de w. Lorsque la simulation de N sur un mot w arrête, N' entre dans une sous-routine qui écrit w sur la ligne -1 et passe dans l'état q_e ; ensuite la ligne -1 est effacée tandis que celle sur laquelle N' a simulé le calcul de N sur w sera désormais ignorée.

Chapitre III Indécidabilité

On démontre qu'il existe des limites à la puissance des machines de Turing et du calcul algorithmique en général.

1 Préliminaires

On commence par définir le problème et par lui donner une première réponse ; comme celle-ci utilise un argument non constructif, on ne s'en satisfera pas.

La "question fondamentale" posée au chapitre I se reformule ainsi :

qu'est-il possible de faire avec une machine de Turing ?

Une fois défini ce modèle de calcul, on peut reformuler cette question avec plus de précision, de manière à pouvoir tirer des conclusions pratiques une fois qu'on y aura répondu :

à quelles questions est-il possible de répondre avec une machine de Turing ?

Dans le contexte de la théorie des langages formels, chacune de ces "questions" peut se traduire sous la forme d'un test concernant l'appartenance d'un mot $w \in \Sigma^*$ à un langage donné $L \subset \Sigma^*$, pour Σ un alphabet approprié. La "réponse" tient alors en un seul bit, et par souci de réalisme, on exige que celui-ci soit produit au terme d'un calcul de longueur finie. En utilisant les définitions du chapitre précédent, on peut reformuler la "question fondamentale" en termes plus "techniques" :

y a-t-il des langages qui ne sont pas décidables ?

Il est légitime de se poser aussi la question :

y a-t-il des langages qui ne sont pas récursivements énumérables ?

On peut répondre à la seconde question avec un argument basé sur le lemme suivant.

Lemme 3.1 Cardinalité de quelques ensembles infinis, pour un alphabet fini quelconque Σ.

i. L'ensemble Σ^* de tous les mots sur Σ est infini dénombrable.

ii. L'ensemble $\mathcal{P}(\Sigma^*)$ de tous les langages sur Σ est infini indénombrable.

iii. L'ensemble $\mathsf{RE}(\Sigma)$ de tous les langages récursivements énumérables sur Σ est infini dénombrable.

iv. Un ensemble infini dénombrable contient strictement moins d'éléments qu'un ensemble infini indénombrable.

La preuve des deux premiers items consiste à établir une bijection entre Σ^* et $\mathbb{N}$ d'une part, et entre $\mathcal{P}(\Sigma^*)$ et $\mathbb{R}$ d'autre part. Pour le troisième item, on procède comme ceci :

1. toute machine de Turing à données provenant de Σ^* est spécifiable par une description de longueur finie écrite avec l'alphabet Σ ;
2. par conséquent, il n'y a pas plus de ces machines de Turing qu'il n'y a de mots dans Σ^* ;
3. par définition, tout langage de $\mathsf{RE}(\Sigma)$ est spécifié par au moins une machine de Turing qui l'accepte ;
4. par conséquent, il n'y a pas plus de langages récursivements énumérables sur Σ qu'il n'y a de machines de Turing à données provenant de Σ^*.

Le dernier item du lemme est un résultat célèbre de Cantor. On remarque que la preuve ne fait qu'affirmer l'existence de langages dans $\mathcal{P}(\Sigma^*) \setminus \mathsf{RE}(\Sigma)$ mais ne donne aucune indication sur la manière d'en trouver un spécimen : on parle d'une preuve non constructive.

2 Le problème de l'arrêt

On commence par montrer qu'il est possible de répondre d'un seul coup aux deux questions posées dans la section précédente.

Lemme 3.2 S'il existe un langage récursivement énumérable qui n'est pas décidable, alors il existe un langage qui n'est pas récursivement énumérable.

Cet énoncé est une conséquence immédiate du fait que $L \subset \Sigma^*$ est décidable si et seulement si L et son complément $\overline{L} = \Sigma^* \setminus L$ sont tous les deux récursivement énumérables.

Définition 3.3 Soit M une machine de Turing à données provenant de Σ^*. Son *encodage* $\varepsilon(M)$ est une chaîne de Σ^* qui satisfait les propriétés suivantes :

i. si M et N sont deux machines différentes, alors $\varepsilon(M) \neq \varepsilon(N)$;

ii. l'ensemble des mots $w \in \Sigma^*$ tels que $\exists\, M : w = \varepsilon(M)$ est un langage décidable ;

iii. il existe une machine U, dite *machine de Turing universelle*, dont la donnée est de la forme $u\square v$, avec u et v deux mots de Σ^* et $\square \notin \Sigma$, et dont le travail consiste à :
   1. tester si u est un encodage
   2. si ce n'est pas le cas, boucler à l'infini ; sinon, simuler exactement le travail de M sur la donnée v, où M est la machine pour laquelle $u = \varepsilon(M)$.

◊ Exercice 3.1 Vérifier qu'il existe des fonctions d'encodage qui satisfont cette définition ; en particulier, décrire en détail comment fonctionne une machine de Turing universelle.

Définition 3.4 Le *problème de l'arrêt* est le langage $L_{\mathtt{halt}}$ des mots x qui satisfont les deux conditions

i. $x = \varepsilon(M)\square w$ pour M une machine de Turing et w une donnée valide pour M,

ii. le calcul de M sur la donnée w arrête : $w \notin L_\infty(M)$.

Théorème 3.5 (*Turing, 1936*) Le langage $L_{\texttt{halt}}$ est récursivement énumérable et indécidable.

Démonstration.
Il est facile de vérifier que ce langage appartient à RE. On démontre l'indécidabilité en définissant un sous-ensemble de $L_{\texttt{halt}}$: $H(1) = \{\ \varepsilon(M) \mid M \text{ arrête sur la donnée } \varepsilon(M)\ \}$, pour lequel on montre que (i) si $L_{\texttt{halt}}$ est décidable alors $H(1)$ l'est aussi, et (ii) $H(1)$ est indécidable ; l'indécidabilité de $L_{\texttt{halt}}$ vient alors par transposition. On démontre (ii) par l'absurde. Supposons en effet que $H(1) \in \texttt{Dec}$; il existe alors une machine N qui arrête sur le complément de $H(1)$, et on a :

$$\begin{array}{rcl} N \text{ arrête sur } \varepsilon(N) & \texttt{ssi} & \varepsilon(N) \notin H(1) \quad \texttt{ssi} \quad N \text{ n'arrête pas sur } \varepsilon(N), \text{ et} \\ N \text{ n'arrête pas sur } \varepsilon(N) & \texttt{ssi} & \varepsilon(N) \in H(1) \quad \texttt{ssi} \quad N \text{ arrête sur } \varepsilon(N), \end{array}$$

ce qui amène à conclure que N ne peut pas exister.

3 Problèmes indécidables

Il existe de très nombreux problèmes indécidables originant de plusieurs domaines différents des mathématiques. Nous en présentons ici quelques-uns en montrant comment prouver qu'ils sont indécidables.

3.1 Problèmes concernant les machines de Turing

Définition 3.6 Problèmes fondamentaux concernant les machines de Turing.

- Le *problème de l'arrêt sur le mot vide* est le langage $L_{\texttt{movi}}$ des mots x qui satisfont les deux conditions
  - i. $x = \varepsilon(M)$ pour M une machine de Turing,
  - ii. quand M reçoit le mot vide λ comme donnée, le calcul de M arrête : $\lambda \notin L_\infty(M)$.

  Ce problème est une restriction du problème de l'arrêt dans laquelle la donnée est une constante et la machine une variable.
- Le *problème de l'arrêt pour la machine universelle* est le langage $L_{\texttt{univ}}$ des mots x qui satisfont les deux conditions
  - i. $x = \varepsilon(M)\square w$ pour M une machine de Turing et w une donnée valide pour M,
  - ii. le calcul de la machine M sur la donnée w arrête : $w \notin L_\infty(M)$.

  Ce problème est une restriction du problème de l'arrêt dans laquelle la donnée est une variable et la machine une constante.
- Le *problème du langage vide* est le langage $L_{\texttt{vide}}$ des mots x qui satisfont les conditions
  - i. $x = \varepsilon(M)$ pour M une machine de Turing,
  - ii. M n'arrête sur aucune donnée : $L_\infty(M) = \Sigma^*$.

Théorème 3.7 Les langages $L_{\texttt{movi}}$, $L_{\texttt{univ}}$ et $L_{\texttt{vide}}$ sont tous les trois indécidables. De plus, les trois langages $L_{\texttt{movi}}$, $L_{\texttt{univ}}$ et $\overline{L_{\texttt{vide}}}$ sont récursivement énumérables. [1]

1. $L_{\texttt{vide}}$ est le complément d'un langage de RE ; c'est un exemple de langage de la classe $\texttt{coRE} = \{\ K \mid \overline{K} \in \texttt{RE}\ \}$.

Pour faire des preuves d'indécidabilité pour des langages comme les trois ci-dessus, on utilise habituellement la technique dite de la *réduction* qui, informellement, consiste à montrer que tester si une donnée appartient à un langage L est au moins aussi ardu (ou si on préfère, exige au moins autant de travail) que tester si un mot est dans $L_{\texttt{halt}}$.

Définition 3.8 Soient $K \subset \Sigma^*$ et $L \subset \Gamma^*$ deux langages. On dit que K est *réductible à* L s'il existe une fonction $f : \Sigma^* \to \Gamma^*$ calculable au sens de Turing qui satisfait les conditions

i. f est une fonction totale, c'est à dire qu'il existe une machine M_f qui pour tout $x \in \Sigma^*$, calcule la valeur de $f(x)$ en un temps fini ;

ii. pour tout $x \in \Sigma^*$, on a $x \in K$ ssi $f(x) \in L$.

On dénote par $K \leq L$ la relation "K est réductible à L".

L'intuition est la suivante : on sait que tester l'appartenance dans K d'un mot de Σ^* est un travail ardu et on veut savoir ce qu'il en est pour L. On construit alors une fonction qui va prendre les données pour l'appartenance à K et les *encoder* sous la forme de données pour l'appartenance à L et qui aura les propriétés spécifiées par la définition. On pourra alors procéder en trois étapes :

1. prendre la donnée $x \in \Sigma^*$ et la traduire en une donnée $f(x) \in \Gamma^*$;
2. tester si $f(x)$ appartient à L ;
3. retourner la réponse fournie par ce test : comme on a $x \in K$ ssi $f(x) \in L$, il n'y a aucun autre travail à faire.

Évidemment, si calculer $f(x)$ est en soi un travail ardu, le détour par L n'a aucun intérêt. Par contre, si on sait d'avance que calculer $f(x)$ est strictement plus facile que décider si $x \in K$, alors on peut appliquer le raisonnement suivant :

1. on sait que tester l'appartenance au langage K exige beaucoup de travail, quelle que soit la méthode utilisée ;
2. alors en particulier, la méthode décrite ci-dessus contient nécessairement au moins une étape ardue ;
3. on sait que calculer la valeur de la fonction f est facile ; on sait aussi que la troisième étape ne nécessite aucun travail ;
4. on conclut que la seconde étape, le test pour $f(x) \in L$, exige beaucoup de travail ; donc, tester l'appartenance au langage L est un problème ardu.

Dans le cas qui nous intéresse, "ardu" veut dire "indécidable" et "exiger beaucoup de travail" signifie "exiger un calcul de longueur infinie". Alors il est clair qu'en comparaison, "calculable en un temps fini" est "facile".

En pratique, pour faire une preuve par réduction de K à L, il faut et il suffit de donner la définition d'une fonction f appropriée et de démontrer qu'elle satisfait les deux conditions énoncées à la définition 3.8.

◊ Exercice 3.2 Démontrer les propositions suivantes.

1. Si $K \leq L$ et L est décidable, alors K est décidable.
2. La relation $\leq$ est transitive.
3. Si $K \leq L$ et L est dans `RE`, alors K est dans `RE`.
4. Si $K \leq L$ et L est régulier, alors K n'est pas nécessairement régulier.

$\Diamond$ Exercice 3.3 Démontrer le théorème 3.7.

Étant donnée la spécification d'un langage sous la forme d'une machine de Turing qui l'accepte, il est normal de vouloir savoir quelles sont les propriétés de ce langage, par exemple s'il appartient à la classe `Reg` des langages réguliers. Le théorème suivant montre qu'il n'existe pas d'algorithme universel pour répondre à cette question.

Théorème 3.9 (*Rice, 1953*) Le problème suivant est indécidable : étant données une machine de Turing M et une partition nontriviale de $\mathcal{P}(\Sigma^*)$, c'est à dire une classe de langages K et son complément, déterminer si le langage accepté par M appartient à K.

3.2 Grammaires et systèmes de réécriture

Il existe une généralisation naturelle pour la notion de grammaire hors-contexte ; on montre qu'elle permet de définir exactement la classe `RE`.

Définition 3.10 Une *grammaire généralisée* est un quadruplet $G = (V, \Sigma, P, S)$ où
V est l'ensemble des symboles non-terminaux,
Σ est celui des symboles terminaux,
$P \subset [(V \cup \Sigma)^* V (V \cup \Sigma)^*] \times (V \cup \Sigma)^*$ est l'ensemble des productions et
$S \in V$ est la variable initiale ;
les trois ensembles V, Σ et P sont finis. On définit les notions de dérivation, de mot et de langage engendrés par la grammaire exactement comme on l'a fait pour les grammaires hors-contexte.

Théorème 3.11 Un langage est récursivement énumérable ssi il existe une grammaire généralisée qui l'engendre.

Théorème 3.12 Les problèmes suivants sont indécidables.

i. Étant donnés une grammaire G et un mot $w \in \Sigma^*$, déterminer si w est dérivable à partir du symbole initial avec les productions de G.

ii. Étant donnée une grammaire G, déterminer si le mot vide λ est dérivable à partir du symbole initial avec les productions de G.

iii. Étant donnée une grammaire G, déterminer si le langage $L(G)$ des mots dérivables à partir du symbole initial avec les productions de G est différent de l'ensemble vide.

La preuve se fait relativement aisément par réduction et consiste à encoder dans les productions d'une grammaire la spécification d'une machine de Turing.

Définition 3.13 Un *système de réécriture* est un couple $S = (\Sigma, R)$ où Σ est un alphabet fini et $R \subset \Sigma^+ \times \Sigma^*$ est l'ensemble des *règles de réécriture*.
On établit dans Σ^* une relation de *dérivabilité en une étape* définie par $w \to_S w'$ ssi $\exists (x, y) \in R$: $[w = uxv \wedge w' = uyv]$. Avec $\overset{*}{\to}_S$ la fermeture réflexive et transitive de cette relation, on associe à chaque mot $w \in \Sigma^*$ les deux langages

$\Delta_S(w) = \{\, v \in \Sigma^* \mid w \overset{*}{\to}_S v \,\}$ l'ensemble des "descendants" de w,
$\nabla_S(w) = \{\, v \in \Sigma^* \mid v \overset{*}{\to}_S w \,\}$ l'ensemble des "ancêtres" de w.

On dit qu'un langage $L \subset \Sigma^*$ est *spécifié par le système de réécriture* S lorsque $L = \Delta_S(w)$ pour un mot donné $w \in \Sigma^*$, ou bien lorsque $L = \nabla_S(w)$ pour un $w \in \Sigma^*$. On dira aussi qu'un langage est spécifiable par système de réécriture lorsqu'il existe un système par lequel ce langage est spécifié.

Théorème 3.14 Un langage est spécifiable par système de réécriture ssi il est récursivement énumérable.

Théorème 3.15 Les problèmes suivants sont indécidables.

i. Étant donnés un système S et deux mots $v, w \in \Sigma^*$, déterminer si $v \in \Delta_S(w)$.
ii. Étant donnés un système S et deux mots $v, w \in \Sigma^*$, déterminer si $v \in \nabla_S(w)$.
iii. Étant donnés un système S et un mot $w \in \Sigma^*$, déterminer si $\lambda \in \Delta_S(w)$.

3.3 Problème de correspondance de Post

Il s'agit d'un exemple classique de problème indécidable.

Définition 3.16 Le *problème de correspondance de Post* (PCP) a pour donnée un ensemble fini P de paires $p_i = (u_i, v_i)$ de mots $u_i, v_i \in \Sigma^*$ et consiste à demander s'il existe une suite finie de paires, $p_1, \ldots, p_n$ qui satisfait la condition $u_1 \cdots u_n = v_1 \cdots v_n$.

Théorème 3.17 Le problème de correspondance de Post est indécidable.

4 Relativisation

On peut croire que l'indécidabilité pourrait représenter le degré ultime de difficulté qu'un problème puisse atteindre. Nous allons montrer qu'il n'en est rien : à partir du problème de l'arrêt, il est possible de construire une hiérarchie infinie de niveaux de complexité.

Définition 3.18 Une *machine de Turing avec oracle pour le langage* A est une machine M^A possédant, en plus de ses caractéristiques habituelles, un ruban spécial appelé "ruban-requête" ainsi qu'un "état-requête" dénoté q_r. La machine M^A fonctionne comme une machine ordinaire tant qu'elle n'entre pas dans l'état q_r ; lorsque cela arrive, le contenu x du ruban-requête est

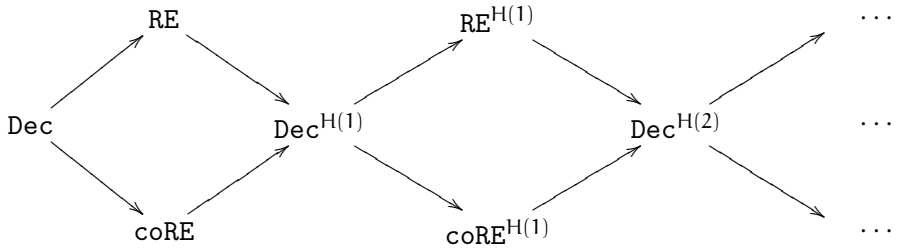

FIGURE 3.1 : Hiérarchie de classes de langages au-dessus de Dec

effacé et remplacé par le symbole 1 si $x \in A$, et par 0 si $x \notin A$; le calcul se poursuit normalement jusqu'au prochain passage de M^A dans l'état q_r, ou bien jusqu'à l'arrêt normal de la machine.

La machine M^A utilise le ruban-requête pour écrire une question à soumettre à un oracle qui répondra en une étape de calcul. Le recours à l'oracle permet de voir jusqu'à quel point on peut augmenter la puissance de calcul d'une machine ordinaire en permettant de tester aussi l'appartenance à un langage indécidable A.

Définition 3.19 Soit $A \subset \Sigma^*$. Un langage $L \subset \Sigma^*$ est *récursivement énumérable relativement à* A ssi il existe une machine de Turing avec oracle pour A qui accepte L. De même, L est *décidable relativement à* A ssi il existe une machine de Turing avec oracle pour A qui décide L. Ceci définit deux classes de langages qu'on dénotera respectivement par $\mathtt{RE}^A$ et $\mathtt{Dec}^A$.

On définit maintenant une suite infinie de langages $H(i)$, $i \geq 1$:

$H(1) = \{\ \varepsilon(M) \mid M \text{ arrête sur la donnée } \varepsilon(M)\ \}$ et

$H(i+1) = \{\ \varepsilon(M^{H(i)}) \mid M^{H(i)} \text{arrête sur la donnée } \varepsilon(M^{H(i)})\ \}$.

Pour un langage $L \subset \Sigma^*$ donné, on peut se demander où L se situe par rapport à ces langages, et plus précisément se demander quels sont les nombres $\max\{i : H(i) \leq L\}$ et $\min\{j : L \leq H(j)\}$. La question sera d'autant plus pertinente que les langages $H(i)$ définissent une *hiérarchie infinie* de degrés de difficulté.

Théorème 3.20 Pour tout $i \geq 2$ on a $\mathtt{Dec}^{H(i-1)} \subset \mathtt{RE}^{H(i-1)} \subset \mathtt{Dec}^{H(i)} \subset \mathtt{RE}^{H(i)}$.

Démonstration.
(1) On commence par prouver les inclusions $\mathtt{Dec}^{H(i-1)} \subset \mathtt{RE}^{H(i-1)}$ et $\mathtt{Dec}^{H(i)} \subset \mathtt{RE}^{H(i)}$; en fait, il suffit de vérifier que ces inclusions sont strictes. Pour ce faire, on montre que $H(i)$, qui appartient à $\mathtt{RE}^{H(i-1)}$, est indécidable relativement à $H(i-1)$. La preuve fonctionne par l'absurde et suit la même logique que celle qu'on a vue pour le problème de l'arrêt. Supposons en effet que $H(i) \in \mathtt{Dec}^{H(i-1)}$: il existe alors une machine $N^{H(i-1)}$ qui arrête sur le complément de $H(i)$, et en appliquant la même logique que pour le théorème 3.5, on arrive à une contradiction.

(2) Il reste à montrer l'inclusion $\mathtt{RE}^{\mathsf{H}(\mathtt{i}-1)} \subset \mathtt{Dec}^{\mathsf{H}(\mathtt{i})}$. On remarque qu'il suffit de prouver $\mathtt{RE}^{\mathsf{H}(\mathtt{i}-1)} \subseteq \mathtt{Dec}^{\mathsf{H}(\mathtt{i})}$, parce que le fait que $\mathtt{Dec}^{\mathsf{H}(\mathtt{i})}$ est fermée sous la complémentation et que $\mathtt{RE}^{\mathsf{H}(\mathtt{i}-1)}$ ne l'est pas impliquera que l'inclusion sera stricte.
La démonstration consiste à montrer que tout langage $\mathsf{L} \in \mathtt{RE}^{\mathsf{H}(\mathtt{i}-1)}$ peut être réduit à $\mathsf{H}(\mathtt{i})$. Soit $\mathsf{M}^{\mathsf{H}(\mathtt{i}-1)}$ une machine qui arrête sur L et w une donnée pour $\mathsf{M}^{\mathsf{H}(\mathtt{i}-1)}$. Il existe une procédure qui, à partir de la chaîne w, construit l'encodage d'une machine $\mathsf{N}_w^{\mathsf{H}(\mathtt{i}-1)}$, telle que (1) cet encodage contient $\varepsilon(\mathsf{M}^{\mathsf{H}(\mathtt{i}-1)})\square w$ comme facteur, et (2) l'algorithme de $\mathsf{N}_w^{\mathsf{H}(\mathtt{i}-1)}$ fonctionne en deux étapes :

i. $\mathsf{N}_w^{\mathsf{H}(\mathtt{i}-1)}$ vérifie si sa donnée contient $\varepsilon(\mathsf{M}^{\mathsf{H}(\mathtt{i}-1)})\square w$ comme facteur ;
ii. si c'est le cas, alors elle simule le travail de $\mathsf{M}^{\mathsf{H}(\mathtt{i}-1)}$ sur w ; sinon, elle boucle à l'infini.

On vérifie que si on soumet $\varepsilon(\mathsf{N}_w^{\mathsf{H}(\mathtt{i}-1)})$ comme donnée à $\mathsf{N}_w^{\mathsf{H}(\mathtt{i}-1)}$, le calcul arrête (c'est à dire : $\varepsilon(\mathsf{N}_w^{\mathsf{H}(\mathtt{i}-1)}) \in \mathsf{H}(\mathtt{i})$) si et seulement si $\mathsf{M}^{\mathsf{H}(\mathtt{i}-1)}$ arrête sur w, autrement dit, $w \in \mathsf{L}$. □

Chapitre IV Les fonctions récursives

On se bornera ici à fixer le vocabulaire et les notations, à énoncer les principaux théorèmes et à traiter quelques notions complémentaires.

1 Définitions

Définition 4.1 Fonctions de base :

la fonction constante nulle à k arguments, $k \geq 1$, $Z^k : \mathbb{N}^k \to \mathbb{N}$;
la fonction constante à k arguments, $k \geq 1$, et à valeur $n \in \mathbb{N}$, $C^k_n : \mathbb{N}^k \to \mathbb{N}$;
la fonction de projection à k arguments, $k \geq 1$, sur le i^{eme} argument, $P^k_i : \mathbb{N}^k \to \mathbb{N}$;
$s(x)$ et $pred(x) : \mathbb{N} \to \mathbb{N}$ les fonctions successeur et prédécesseur.

Définition 4.2 Schémas de construction :

composition, $f(x_1, \ldots, x_k) = g(h_1(x_1, \ldots, x_k), \ldots, h_m(x_1, \ldots, x_k))$;

récurrence simple, $\begin{cases} f(x_1, \ldots, x_k, 0) = g(x_1, \ldots, x_k) \\ f(x_1, \ldots, x_k, s(y)) = h(x_1, \ldots, x_k, y, f(x_1, \ldots, x_k, y)) \end{cases}$;

addition finie, $f(x_1, \ldots, x_k, y) = \begin{cases} 0 & \text{si } y = 0 \\ \sum_{z<y} g(x_1, \ldots, x_k, z) & \text{sinon} \end{cases}$;

multiplication finie, $f(x_1, \ldots, x_k, y) = \begin{cases} 1 & \text{si } y = 0 \\ \prod_{z<y} g(x_1, \ldots, x_k, z) & \text{sinon} \end{cases}$;

valeur minimum, $f(x_1, \ldots, x_k, y) = \min\{\, g(x_1, \ldots, x_k, z) \mid z \leq y \,\}$;

valeur maximum, $f(x_1, \ldots, x_k, y) = \max\{\, g(x_1, \ldots, x_k, z) \mid z \leq y \,\}$;

minimum effectif[2], $f(x_1, \ldots, x_k) = \begin{cases} 0 & \text{si } (\forall y)[\, g(x_1, \ldots, x_k, y) > 0 \,] \\ \min\{\, y \in \mathbb{N} \mid g(x_1, \ldots, x_k, y) = 0 \,\} & \text{sinon} \end{cases}$.

Définition 4.3 Les *fonctions récursives primitives* sont définies récursivement comme ceci :

i. la fonction successeur $s(x)$ et pour tout k, pour tout $i \leq k$, les fonctions Z^k et P^k_i sont récursives primitives ;
ii. toute fonction construite à partir de fonctions récursives primitives en utilisant le schéma de composition ou le schéma de récurrence simple est elle aussi récursive primitive ;
iii. clause limitative.

On dénote par $\mathcal{F}_P$ la classe des fonctions récursives primitives.

Exemple 4.4 On montre que les fonctions suivantes sont récursives primitives :

1. les fonctions constantes C^k_n ;

2. Noter que dans le cas $f(x_1, \ldots, x_k) = y > 0$, il est supposé que $g(x_1, \ldots, x_k, z)$ est défini pour tout $z < y$.

2. la fonction prédécesseur $\texttt{pred}(x)$;
3. F_+, la fonction somme.

Pour les fonctions constantes C_n^k, on procède par induction sur n, avec pour base $C_0^k = Z^k$ et pour étape la définition de $C_{n+1}^k(x_1, \ldots, x_k) = s(C_n^k(x_1, \ldots, x_k))$ par le schéma de composition.
Pour la fonction prédécesseur $\texttt{pred}(x)$, on utilise les schémas de composition : $\texttt{pred}(x) = f(x, x)$, puis celui de récurrence simple : $f(x, 0) = Z^1(x)$, $f(x, s(y)) = P_2^3(x, y, f(x, y))$. Le passage par la fonction f permet d'éviter d'avoir à définir une "fonction constante nulle à 0 argument" à la base de l'induction.
Pour la fonction somme à deux arguments $F_+^2(x, y)$, on utilise la définition récursive de l'addition $F_+^2(x, 0) = P_1^1(x)$ et $F_+^2(x, s(y)) = s(P_3^3(x, y, F_+^2(x, y))$, en combinant les schémas de composition et de récurrence simple. Ensuite, la fonction somme à k arguments $F_+^k(x_1, \ldots, x_k)$ est définie par composition :
$F_+^k(x_1, \ldots, x_k) = F_+^2(P_k^k(x_1, \ldots, x_k), F_+^{k-1}(P_1^k(x_1, \ldots, x_k), \ldots, P_{k-1}^k(x_1, \ldots, x_k)))$.

L'exemple montre que les notations deviennent vite très lourdes. Aussi se permettra-t-on de

- utiliser les notations habituelles pour l'addition et la multiplication,
- se passer des fonctions de projection lorsque leur liste d'arguments est évidente, en particulier quand on utilise les schémas de composition et de récurrence simple.

Par exemple, la fonction somme pourra se définir comme ceci :
$F_+^2(x, 0) = P_1^1(x)$ et $F_+^2(x, s(y)) = s(F_+^2(x, y))$; $\quad F_+^k(x_1, \ldots, x_k) = F_+^2(x_k, F_+^{k-1}(x_1, \ldots, x_{k-1}))$.

Exemple 4.5 Définition de la fonction produit $F_\times^k$.
$F_\times^2(x, 0) = Z^1(x)$ et $F_\times^2(x, s(y)) = x + F_\times^2(x, y)$;
$F_\times^k(x_1, \ldots, x_k) = F_\times^2(x_k, F_\times^{k-1}(x_1, \ldots, x_{k-1}))$.

Pour définir une fonction $f : x \mapsto x^2$ on pourra se contenter d'écrire $f(x) = x \cdot x$, en mentionnant qu'on utilise pour cela le schéma de composition.

◊ Exercice 4.1 Montrer que les fonctions suivantes sont récursives primitives :

1. F_{dp} la différence pointée ;
2. $\texttt{sg}$ et $\overline{\texttt{sg}}$ les signum et signum inverse ;
3. $\texttt{reste}(x, y) = x \bmod y$ le reste de la division entière ;
4. $\texttt{premier}(n)$, la fonction caractéristique du prédicat qui prend la valeur $\texttt{vrai}$ ssi $n \geq 2$ et n est un nombre premier ;
5. $P : \mathbb{N} \to \mathbb{N}$, telle que $P(n)$ est le n^e nombre premier, selon la convention $P(0) = 2$, $P(1) = 3$, $P(2) = 5$, etc. ;
6. $\beta_k : \mathbb{N} \to \mathbb{N}$, $0 \leq k$, telle que $\beta(0) = 0$ et pour $n > 0$, $\beta_k(n)$ est l'exposant de $P(k)$ dans la décomposition de n en puissances de nombres premiers.

◊ EXERCICE 4.2 Démontrer que la classe $\mathcal{F}_P$ est fermée sous chacun des schémas de construction suivants :
1. addition finie ; 2. multiplication finie ; 3. valeur minimum ; 4. valeur maximum.

◊ EXERCICE 4.3 Démontrer que la classe $\mathcal{F}_P$ est fermée sous le schéma de récurrence conditionnelle, défini par

$$\begin{cases} f(x_1,\ldots,x_k,0) = g(x_1,\ldots,x_k) \\ f(x_1,\ldots,x_k,s(y)) = \begin{cases} h_1(x_1,\ldots,x_k,y,f(x_1,\ldots,x_k,y)) & \text{si} \quad p_1(x_1,\ldots,x_k,y)=1; \\ \cdots \qquad\qquad\qquad\qquad\quad \cdots & \cdots \\ h_\ell(x_1,\ldots,x_k,y,f(x_1,\ldots,x_k,y)) & \text{si} \quad p_\ell(x_1,\ldots,x_k,y)=1. \end{cases} \end{cases}$$

Ici, $p_1,\ldots,p_\ell$ satisfont la condition $p_1(x_1,\ldots,x_k,y)+\cdots+p_\ell(x_1,\ldots,x_k,y)=1$ et peuvent être vues comme les fonctions caractéristiques de prédicats mutuellement exclusifs.

Définition 4.6 Les *fonctions récursives* (appelées aussi récursives générales) sont définies récursivement comme ceci :

i. la fonction successeur $s(x)$ et pour tout k, pour tout $i \le k$, les fonctions Z^k et P_i^k sont récursives primitives ;

ii. toute fonction construite à partir de fonctions récursives en utilisant le schéma de composition, le schéma de récurrence simple et le schéma de minimum effectif, est elle aussi récursive ;

iii. clause limitative.

On dénote par $\mathcal{F}_G$ la classe des fonctions récursives.

Définition 4.7 Fonction d'Ackermann : sa définition se fait en deux étapes. On construit d'abord une suite infinie de fonctions a_i, $i \ge 1$:

$$a_1(x,0)=x \quad \text{et} \quad a_1(x,s(y))=s(a_1(x,y)),$$
$$a_2(x,0)=0 \quad \text{et} \quad a_2(x,s(y))=a_1(x,a_2(x,y)), \text{ et}$$
$$\text{pour } i\ge 3, \quad a_i(x,0)=1 \text{ et } \quad a_i(x,s(y))=a_{i-1}(x,a_i(x,y)).$$

La fonction d'Ackermann [3] est alors donnée par : $A(0)=1$ et pour $x\ge 1$, $A(x)=a_x(x,x)$.

◊ EXERCICE 4.4 Donner une expression analytique pour chaque $a_i(x,y)$, $i \le 4$.

Théorème 4.8 La classe $\mathcal{F}_P$ est un sous-ensemble strict de $\mathcal{F}_G$.

L'inclusion au sens large vient immédiatement des définitions ; la suite de la démonstration consiste à prouver que la fonction d'Ackermann augmente plus vite que n'importe quelle fonction récursive primitive : pour toute fonction à une variable $f \in \mathcal{F}_P$, il existe $i \ge 1$ tel que $\forall\, n \ge i : A(n) > f(n)$. La fonction A est donc dans $\mathcal{F}_G$ mais pas dans $\mathcal{F}_P$.

3. En fait, ceci est une parmi plusieurs versions possibles de cette fonction.

2 Thèse de Turing

On a déjà vu une classe de fonctions, celles qui sont calculables au sens de Turing (voir la définition 2.8). La thèse de Turing est le théorème suivant.

Théorème 4.9 La classe des fonctions calculables au sens de Turing coïncide avec celle des fonctions récursives.

Toute fonction récursive est calculable au sens de Turing : la preuve se fait en montrant comment simuler les schémas de construction avec des machines de Turing. Pour l'autre direction, qui affirme que toute fonction calculable au sens de Turing est récursive, on donnera une preuve indirecte au dernier chapitre ; il existe aussi une preuve directe dont la description détaillée se trouve dans le chapitre 13 du livre de Sudkamp.

3 Hiérarchies de Grzegorczyk

On démontre que la fonction d'Ackermann n'est que la base d'une hiérarchie de fonctions récursives qui augmentent toujours plus rapidement.

A la définition 4.7, on a construit la famille de fonctions $\{a_1, a_2, a_3, \ldots\} \cup \{A\}$; on utilisait pour cela de façon cruciale la fonction successeur. On reprend l'exercice en remplaçant la fonction s par A.

Définition 4.10 Construction d'une famille de fonctions $\{b_1, b_2, b_3, \ldots\} \cup \{B\}$. On commence par $b_1(x, 0) = x$ et $b_1(x, s(y)) = A(b_1(x, y))$, et ensuite pour $i \geq 2$ on définit $b_i(x, 0) = 1$ et $b_i(x, s(y)) = b_{i-1}(x, b_i(x, y))$.
La fonction B est ensuite donnée par : $B(0) = 1$ et pour $x \geq 1$, $B(x) = b_x(x, x)$.

On peut montrer que B domine A. On peut bien entendu reprendre la définition 4.10 en utilisant B au lieu de A, etc. On obtient ainsi une famille infinie de fonctions $\phi_0 = s$, $\phi_1 = A$, $\phi_2 = B$, ..., où chaque ϕ_j augmentera strictement plus vite que ϕ_{j-1}.
A partir de cette famille, on peut maintenant construire une fonction $\Phi : x \mapsto \phi_x(x)$, c'est à dire telle que $\Phi(0) = s(0) = 1$, $\Phi(1) = A(1)$, $\Phi(2) = B(2)$, etc.
Évidemment, il est tout-à-fait possible de reprendre le processus, en définissant d'abord une suite infinie de fonctions p_i, $i \geq 1$, telle que $p_1(x, 0) = x$ et $p_1(x, s(y)) = \Phi(p_1(x, y))$, et en définissant les fonctions p_i, $i \geq 2$, comme en 4.10. On aura compris qu'il n'y a pas de limite à ce processus.

Bibliographie

[1] S. Arora et B. Barak, *Computational complexity, a modern approach,* Cambridge (2009).

[2] J. Balcázar, J. Diáz, J. Gabarró, *Structural Complexity I* et *Structural Complexity II,* Springer-Verlag (1995,1990).

[3] R. Cori, et D. Lascar, *Logique mathématique, cours et exercices* Masson (1993).

[4] M.R. Garey et D.S. Johnson, *Computers and Intractability, A Guide to the Theory of NP-Completeness,* Freeman (1979).

[5] L. Hemaspaandra, M. Ogihara, *The Complexity Theory Companion,* Springer-Verlag (2002).

[6] J. Hopcroft, R. Motwani et J. Ullman, *Introduction Automata Theory, Languages and Computation, 2nd Edition,* Addison-Wesley (2000).

[7] J. Hopcroft et J. Ullman, *Introduction Automata Theory, Languages and Computation,* Addison-Wesley (1979).

[8] H. Lewis et C. Papadimitriou, *Elements of the Theory of Computation, 1st Edition,* Prentice-Hall (1981).

[9] H. Lewis et C. Papadimitriou, *Elements of the Theory of Computation, 2nd Edition,* Prentice-Hall (1998).

[10] P. Linz, *Introduction to Formal Languages and Automata,* Jones and Bartlett (1997).

[11] M. Machtey et P. Young, *An Introduction to the General Theory of Algorithms,* North Holland (1978).

[12] E. Mendelson, *Introduction to Mathematical Logic, 3rd Edition,* Chapman and Hall (1987).

[13] P.G. Odifreddi, *Classical Recursion Theory,* Elsevier (1992) et *Classical Recursion Theory, volume II,* Elsevier (1999).

[14] C. Papadimitriou, *Computational Complexity,* Addison-Wesley (1994).

[15] M. Sipser, *Introduction to the Theory of Computation, 2nd Edition,* PWS (2006).

[16] T. Sudkamp, *Languages and Machines, 2nd Edition,* Addison-Wesley (1997).

[17] I. Wegener, *Complexity Theory,* Springer-Verlag (2005).

[18] P. Wolper, *Introduction à la calculabilité, 3e édition,* Dunod (2006).