Chapitre 8

Calculabilité

Ce chapitre présente les principaux résultats de la calculabilité. En d'autres termes, nous cherchons à comprendre la puissance des programmes informatiques. On y prouve que certains problèmes peuvent se résoudre informatiquement et que certains problèmes ne peuvent pas l'être. L'objectif est d'explorer les limites de la programmation informatique.

En pratique, on peut se dire qu'en informatique on s'intéresse à résoudre des problèmes, en les programmant, par des algorithmes, et que s'intéresser aux problèmes que l'on ne sait pas programmer ne présente que peu d'intérêt. Tout d'abord il convient de comprendre que les problèmes auquels nous allons nous intéresser ne sont pas vraiment des problèmes pour lesquels on ne connaît pas de solution, mais, et c'est en encore beaucoup plus fort, des problèmes pour lesquels on sait qu'il est impossible de produire une solution algorithmique.

Pourquoi s'intéresser à comprendre les problèmes qui ne peuvent pas être résolus ? Premièrement, parce que comprendre qu'un problème ne peut pas être résolu est utile. Cela signifie que le problème doit être simplifié ou modifié pour pouvoir être résolu. Deuxièmement, parce que tous ces résultats sont culturellement très intéressants et permettent de bien mettre en perspective la programmation, et les limites des dispositifs de calculs, ou de l'automatisation de certaines tâches, comme par exemple la vérification.

8.1 Machines universelles

8.1.1 Interpréteurs

Un certain nombre de ces résultats est la conséquence d'un fait simple, mais qui mène à de nombreuses conséquences : on peut programmer *des interpréteurs*, c'est-à-dire des programmes qui prennent en entrée la description d'un autre programme et qui simulent ce programme.

Exemple 8.1 *Un langage comme JAVA est interprèté : un programme JAVA est compilé en une description dans un codage que l'on appelle* bytecode. *Lorsqu'on cherche à*

lancer ce programme, l'interpréteur JAVA simule ce bytecode. Ce principe d'interprétation permet à un programme JAVA de fonctionner sur de nombreuses plates-formes et machines directement : seulement l'interpréteur dépend de la machine sur laquelle on exécute le programme. Le même bytecode peut être lancé sur toutes les machines. C'est en partie ce qui a fait le succès de JAVA : sa portabilité.

La possibilité de programmer des interpréteurs est donc quelque chose d'extrêmement positif.

Cependant, cela mène aussi à de nombreux résultats négatifs ou paradoxaux à propos de l'impossibilité de résoudre informatiquement certains problèmes, même très simples, comme nous allons le voir dans la suite.

Programmer des interpréteurs est possible dans tous les langages de programmation usuels, en particulier même pour les langages aussi rudimentaires que ceux des machines de Turing.

Remarque 8.1 *Nous n'allons pas parler de JAVA dans ce qui suit, mais plutôt de programmes de machines de Turing. Raisonner sur JAVA (ou tout autre langage) ne ferait que compliquer la discussion, sans changer le fond des arguments.*

Commençons par nous persuader que l'on peut réaliser des interpréteurs pour les machines de Turing : dans ce contexte, on appelle cela des *machines de Turing universelles*.

8.1.2 Codage d'une machine de Turing

Il nous faut fixer une représentation des programmes des machines de Turing. Le codage qui suit n'est qu'une convention. Tout autre codage qui garantit la possibilité de décoder (par exemple dans l'esprit du lemme 8.1) ferait l'affaire.

Définition 8.1 (Codage d'une machine de Turing) *Soit M une machine de Turing sur l'alphabet $\Sigma = \{0, 1\}$.*

Selon la définition 7.1Machine de Turingdefinition.7.1, M correspond à un 8-uplet

$$M = (Q, \Sigma, \Gamma, B, \delta, q_0, q_a, q_r) :$$

- *Q est un ensemble fini, dont on peut renommer les états $Q = \{q_1, q_2, \cdots, q_z\}$, avec la convention que $q_1 = q_0$, $q_2 = q_a$, $q_3 = q_r$;*
- *Γ est un ensemble fini, dont on peut renommer les états $\Gamma = \{X_1, X_2, \cdots, X_s\}$, avec la convention que X_s est le symbole B, et que X_1 est le symbole 0 de Σ, et que X_2 est le symbole 1 de Σ.*

Pour $m \in \{\leftarrow, |, \rightarrow\}$, définissons $\langle m \rangle$ de la façon suivante : $\langle \leftarrow \rangle = 1$, $\langle | \rangle = 2$, $\langle \rightarrow \rangle = 3$.

On peut alors coder la fonction de transition δ de la façon suivante : supposons que l'une des règles de transition soit $\delta(q_i, X_j) = (q_k, X_l, m)$: le codage de cette règle est le mot $0^i 1 0^j 1 0^k 1 0^l 1 1^{\langle m \rangle}$ sur l'alphabet $\{0, 1\}$. Observons que puisque tous les entiers i, j, k, l sont non nuls, il n'y pas de 1 consécutif dans ce mot.

Un codage, noté $\langle M \rangle$, de la machine de Turing M est un mot sur l'alphabet $\{0,1\}$ de la forme

$$C_1 11 C_2 11 C_3 \cdots C_{n-1} 11 C_n,$$

où chaque C_i est le codage d'une des règles de transition de δ.

Remarque 8.2 *A une machine de Turing peuvent correspondre plusieurs codages : on peut en particulier permuter les $(C_i)_i$, ou les états, etc....*

Le seul intérêt de ce codage est qu'il est décodable : on peut retrouver chacun des ingrédients de la description d'une machine de Turing à partir du codage de la machine.

Par exemple, si l'on veut retrouver le déplacement m pour une transition donnée :

Lemme 8.1 (Décodage du codage) *On peut construire une machine de Turing M à quatre rubans, telle que si l'on met un codage $\langle M' \rangle$ d'une machine M' sur son premier ruban, 0^i sur son second, et 0^j sur son troisième, M produit sur son quatrième ruban le codage $\langle m \rangle$ du déplacement $m \in \{\leftarrow, |, \rightarrow\}$ tel que $\delta(q_i, X_j) = (q_k, X_l, m)$ où δ est la fonction de transition de la machine de Turing M'.*

Démonstration (principe) : On construit une machine qui parcourt le codage de M' jusqu'à trouver le codage de la transition correspondante, et qui lit dans le codage de cette transition la valeur de m désirée. □

Nous aurons aussi besoin de coder des couples constitués du codage d'une machine de Turing M et d'un mot w sur l'alphabet $\{0,1\}$. Une façon de faire est de définir le codage de ce couple, noté $\langle \langle M \rangle, w \rangle$ par

$$\langle \langle M \rangle, w \rangle = \langle M \rangle 111 w,$$

c'est-à-dire comme le mot obtenu en concaténant le codage de la machine de Turing M, trois fois le symbole 1, puis le mot w. Notre choix de codage d'une machine de Turing ne produisant jamais trois 1 consécutifs, l'idée est qu'on peut alors bien retrouver à partir du mot $\langle M, 1 \rangle 11 w$ ce qui correspond à M et ce qui correspond à w : bref, on peut bien décoder.

8.1.3 Coder des paires, des triplets, etc...

Nous venons de fixer un codage qui fonctionne pour une machine de Turing et un mot, mais on peut faire la remarque suivante : on peut de façon très générale fixer une façon de coder deux mots w_1 et w_2 en un unique mot w, c'est-à-dire de coder un couple de mots, i.e. un élément $\Sigma^* \times \Sigma^*$, par un mot, c'est-à-dire un unique élément de Σ^*, que l'on notera $\langle w_1, w_2 \rangle$.

Comment faire cela ?

Une première façon est de coder deux mots sur Σ par un unique mot sur un alphabet plus grand, de telle sorte que l'on soit capable de retrouver ces mots.

Par exemple, on peut décider de coder les mots $w_1 \in \Sigma^*$ et $w_2 \in \Sigma^*$ par le mot $w_1 \# w_2$ sur l'alphabet $\Sigma \cup \{\#\}$. Une machine de Turing peut alors retrouver à partir de ce mot à la fois w_1 et w_2.

On peut même recoder le mot obtenu en binaire lettre par lettre pour obtenir une façon de coder deux mots sur $\Sigma = \{0,1\}$ par un unique mot sur $\Sigma = \{0,1\}$: par exemple, si $w_1 \# w_2$ s'écrit lettre à lettre $a_1 a_2 \cdots a_n$ sur l'alphabet $\Sigma \cup \{\#\}$, on définit $\langle w1, w_2 \rangle$ comme le mot $e(a_1)e(a_2)\ldots e(a_n)$ où $e(0) = 00$, $e(1) = 01$ et $e(\#) = 10$. Ce codage est encore décodable : à partir de $\langle w_1, w_2 \rangle$, une machine de Turing peut reconstruire w_1 et w_2.

Par la suite, nous noterons $\langle w_1, w_2 \rangle$ le codage de la paire constituée du mot w_1 et du mot w_2.

On observera que les résultats qui suivent ne dépendent pas vraiment du codage utilisé pour les paires : on peut donc coder une paire constitué d'une machine de Turing et d'un mot comme dans la section précédente, ou le considérer comme $\langle \langle M \rangle, w \rangle$, c'est-à-dire le codage du couple constitué du codage de la machine et du mot, indifféremment.

8.1.4 Existence d'une machine de Turing universelle

Ces discussions préliminaires étant faites, on peut se convaincre que l'on peut réaliser un interpréteur, c'est-à-dire ce que l'on appelle *une machine de Turing universelle* dans le contexte des machines de Turing.

Cela donne le théorème suivant :

Théorème 8.1 (Existence d'une machine de Turing universelle) *Il existe une machine de Turing M_{univ} telle, que sur l'entrée $\langle \langle A \rangle, w \rangle$ où :*

1. *$\langle A \rangle$ est le codage d'une machine de Turing A ;*
2. *$w \in \{0,1\}^*$;*

M_{univ} simule la machine de Turing A sur l'entrée w.

Démonstration : On peut facilement se convaincre qu'il existe une machine de Turing M_{univ} à trois rubans telle que si l'on place :

– le codage $\langle A \rangle$ d'une machine de Turing A sur le premier ruban ;

– un mot w sur l'alphabet $\Sigma = \{0,1\}$ sur le second ;

alors M_{univ} simule la machine de Turing A sur l'entrée w en utilisant son troisième ruban.

En effet, la machine M_{univ} simule transition par transition la machine A sur l'entrée w sur son second ruban : M_{univ} utilise le troisième ruban pour y stocker 0^q où q code l'état de la machine A à la transition que l'on est en train de simuler : initialement, ce ruban contient 0, le codage de q_0.

Pour simuler chaque transition de A, M_{univ} lit la lettre X_j en face de sa tête de lecture sur le second ruban, va lire dans le codage $\langle A \rangle$ sur le premier ruban la valeur de q_k, X_l et m, pour la transition $\delta(q_i, X_j) = (q_k, X_l, m)$ de A, où 0^i est le contenu du troisième ruban. M_{univ} écrit alors X_l sur son second ruban, écrit q_k sur son troisième ruban, et déplace la tête de lecture selon le déplacement m sur le second ruban.

Pour prouver le résultat, il suffit alors d'utiliser une machine de Turing avec un unique ruban qui simule la machine à plusieurs rubans précédente, après avoir décodé $\langle A \rangle$ et w à partir de son entrée. □

8.1.5 Premières conséquences

Voici une première utilisation de l'existence d'interpréteurs : la preuve de la proposition 7.3Machines de Turing non-déterministesproposition.7.3, c'est-à-dire la preuve qu'une machine de Turing non déterministe M peut être simulée par une machine de Turing déterministe.

La preuve est la suivante : la relation de transition de la machine non déterministe M est nécessairement de *degré de non determinisme* borné : c'est-à-dire, le nombre

$$r = max_{q \in Q, a \in \Gamma} |\{((q,a),(q',a',m)) \in \delta\}|$$

est fini ($|.|$ désigne le cardinal).

Supposons que pour chaque paire (q,a) on numérote les choix parmi la relation de transition de la machine non déterministe M de 1 à (au plus) r. A ce moment-là, pour décrire les choix non déterministes effectués dans un calcul jusqu'au temps t, il suffit de donner une suite de t nombres entre 1 et (au plus) r.

On peut alors construire une machine de Turing (déterministe) qui simule la machine M de la façon suivante : pour $t = 1, 2, \cdots$, elle énumère toutes les suites de longueur t d'entiers entre 1 et r. Pour chacune de ces suites, elle simule t étapes de la machine M en utilisant les choix donnés par la suite pour résoudre chaque choix non déterministe de M. La machine s'arrête dès qu'elle trouve t et une suite telle que M atteint une configuration acceptante.

8.2 Langages et problèmes décidables

Une fois ces résultats établis sur l'existence de machines universelles (interpréteurs), nous allons dans cette section essentiellement présenter quelques définitions.

Tout d'abord, il nous faut préciser que l'on s'intéresse dorénavant dans ce chapitre uniquement aux problèmes dont la réponse est soit *vraie* soit *fausse*, et ce, essentiellement pour simplifier la discussion : voir la figure 8.1.

8.2.1 Problèmes de décision

Définition 8.2 *Un* problème de décision $\mathcal{P}$ *est la donnée d'un ensemble* E, que l'on appelle l'ensemble des instances*, et d'un sous-ensemble* E^+ *de* E*, que l'on appelle* l'ensemble des instances positives*.*

La question à laquelle on s'intéresse est de construire (si cela est possible) un algorithme qui décide si une instance donnée est positive ou non. On va formuler les problèmes de décision systématiquement sous la forme :

Définition 8.3 (Nom du problème)
Donnée*: Une instance (i.e. un élément de* E*).*
Réponse*: Décider une certaine propriété (i.e. si cet élément est dans* E^+*).*

On peut par exemple considérer les problèmes suivants :

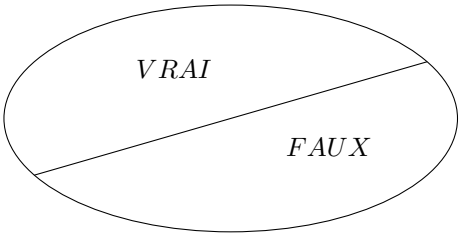

FIGURE 8.1 – Problèmes de décision : dans un problème de décision, on a une propriété qui est soit vraie soit fausse pour chaque instance. L'objectif est de distinguer les instances positives E^+ (où la propriété est vraie) des instances négatives $E \backslash E^+$ (où la propriété est fausse).

Exemple 8.2 ***Définition 8.4*** *(*NOMBRE PREMIER*)*
Donnée*: Un entier naturel n.*
Réponse*: Décider si n est premier.*

Exemple 8.3 ***Définition 8.5*** *(*CODAGE*)*
Donnée*: Un mot w.*
Réponse*: Décider si w correspond au codage $\langle M \rangle$ d'une certaine machine de Turing M.*

Exemple 8.4 ***Définition 8.6*** *(*REACH*)*
Donnée*: Un triplet constitué d'un graphe G, d'un sommet u et d'un sommet v du graphe.*
Réponse*: Décider s'il existe un chemin entre u et v dans le graphe G.*

8.2.2 Problèmes versus Langages

On utilise indifféremment la terminologie problème ou langage dans tout ce qui suit.

Remarque 8.3 (Problèmes vs langages) *Cela découle des considérations suivantes : à un problème (de décision) est associé un langage et réciproquement.*

En effet, à un problème est associé généralement implicitement une fonction de codage (par exemple pour les graphes, une façon de coder les graphes) qui permet de coder les instances, c'est-à-dire les éléments de E, par un mot sur un certain alphabet Σ. On peut donc voir E comme un sous-ensemble de Σ^*, où Σ est un certain alphabet : au problème de décision $\mathcal{P}$, on associe le langage $L(\mathcal{P})$ correspondant à l'ensemble des mots codant une instance de E, qui appartient à E^+ :

$$L(\mathcal{P}) = \{w | w \in E^+\}.$$

Réciproquement, on peut voir tout langage L comme un problème de décision, en le formulant de cette façon :

Définition 8.7 (Problème associé au langage** L**)
***Donnée**: Un mot w.*
***Réponse**: Décider si $w \in L$.*

8.2.3 Langages décidables

On rappelle la notion de langage décidé, qui invite à introduire la notion de langage *décidable*.

Définition 8.8 (Langage décidable) *Un langage $L \subset M^*$ est dit* décidable *s'il est décidé par une certaine machine de Turing.*

Un langage ou un problème décidable est aussi dit *récursif*. Un langage qui n'est pas décidable est dit *indécidable*.

On note R pour la classe des langages et des problèmes *décidables*.

Par exemple :

Proposition 8.1 *Les problèmes de décision* NOMBRE PREMIER, CODAGE *et* REACH *sont décidables.*

La preuve consiste à construire une machine de Turing qui reconnaît respectivement si son entrée est un nombre premier, le codage d'une machine de Turing, où une instance positive de REACH, ce que nous laissons en exercice de programmation élémentaire à notre lecteur.

Exercice 8.1. *Soit A le langage constitué de la seule chaine s où*

$$s = \begin{cases} 0 & \textit{si Dieu n'existe pas} \\ 1 & \textit{si Dieu existe} \end{cases}$$

Est-ce que A est décidable ? Pourquoi ? (la réponse ne dépend pas des convictions religieuses du lecteur).

8.3 Indécidabilité

Dans cette section, nous prouvons un des théorèmes les plus importants philosophiquement dans la théorie de la programmation : l'existence de problèmes qui ne sont pas décidables (i.e. *indécidables*).

8.3.1 Premières considérations

Observons tout d'abord, que cela peut s'établir simplement.

Théorème 8.2 *Il existe des problèmes de décision qui ne sont pas décidables.*

Démonstration : On a vu que l'on pouvait coder une machine de Turing par un mot fini sur l'alphabet $\Sigma = \{0, 1\}$: voir la définition 8.1. Il y a donc un nombre dénombrable de machines de Turing.

Par contre, il y a un nombre non dénombrable de langages sur l'alphabet $\Sigma = \{0, 1\}$: cela peut se prouver en utilisant un argument de diagonalisation comme dans le premier chapitre.

Il y a donc des problèmes qui ne correspondent à aucune machine de Turing (et il y en a même un nombre non dénombrable). □

Le défaut d'une preuve comme celle-là est évidemment qu'elle ne dit rien sur les problèmes en question. Est-ce qu'ils sont ésotériques et d'aucun intérêt sauf pour le théoricien ?

Malheureusement, même des problèmes simples et naturels s'avèrent ne pas pouvoir se résoudre par algorithme.

8.3.2 Est-ce grave ?

Par exemple, dans l'un des problèmes indécidables, on se donne un programme et une spécification de ce que le programme est censé faire (par exemple trier des nombres), et l'on souhaite vérifier que le programme satisfait sa spécification.

On pourrait espérer que le processus de la vérification puisse s'automatiser, c'est-à-dire que l'on puisse construire un algorithme qui vérifie si un programme satisfait sa spécification. Malheureusement, cela est impossible : le problème général de la vérification ne peut pas se résoudre par ordinateur.

On rencontrera d'autres problèmes indécidables dans ce chapitre. Notre objectif est de faire sentir à notre lecteur le type de problèmes qui sont indécidables, et de comprendre les techniques permettant de prouver qu'un problème ne peut pas être résolu informatiquement.

8.3.3 Un premier problème indécidable

On va utiliser un argument de *diagonalisation*, c'est-à-dire un procédé analogue à la diagonalisation de Cantor qui permet de montrer que l'ensemble des parties de $\mathbb{N}$ n'est pas dénombrable : voir le premier chapitre.

Remarque 8.4 *Derrière l'argument précédent sur le fait qu'il y a un nombre non dénombrable de langages sur l'alphabet $\Sigma = \{0, 1\}$ se cachait déjà une diagonalisation. On fait ici une diagonalisation plus explicite, et constructive.*

On appelle *langage universel*, appelé encore *problème de l'arrêt des machines de Turing* , le problème de décision suivant :

Définition 8.9 (HALTING-PROBLEM)
***Donnée**: Le codage $\langle M \rangle$ d'une machine de Turing M et un mot w.*
***Réponse**: Décider si la machine M accepte le mot w.*

Remarque 8.5 *On peut aussi voir ce problème de la façon suivante : on se donne une paire $\langle \langle M \rangle, w \rangle$, où $\langle M \rangle$ est le codage d'une machine de Turing M, et w est un mot, et l'on souhaite décider si la machine M accepte le mot w.*

Théorème 8.3 *Le problème* $\mathrm{HALTING} - \mathrm{PROBLEM}$ *n'est pas décidable.*

Démonstration : On prouve le résultat par un raisonnement par l'absurde. Supposons que $\mathrm{HALTING} - \mathrm{PROBLEM}$ soit décidé par une machine de Turing A.

On construit alors une machine de Turing B qui fonctionne de la façon suivante :
- B prend en entrée un mot $\langle C \rangle$ codant une machine de Turing C ;
- B appelle la machine de Turing A sur la paire $\langle \langle C \rangle, \langle C \rangle \rangle$ (c'est-à-dire sur l'entrée constituée du codage de la machine de Turing C, et du mot w correspondant aussi à ce même codage) ;
- Si la machine de Turing A :
  - accepte ce mot, B refuse ;
  - refuse ce mot, B accepte.

Par construction B termine sur toute entrée.

On montre qu'il y a une contradiction, en appliquant la machine de Turing B sur le mot $\langle B \rangle$, c'est-à-dire sur le mot codant la machine de Turing B.
- Si B accepte $\langle B \rangle$, alors cela signifie, par définition de $\mathrm{HALTING} - \mathrm{PROBLEM}$ et de A, que A accepte $\langle \langle B \rangle, \langle B \rangle \rangle$. Mais si A accepte ce mot, B est construit pour refuser son entrée $\langle B \rangle$. Contradiction.
- Si B refuse $\langle B \rangle$, alors cela signifie, par définition de $\mathrm{HALTING} - \mathrm{PROBLEM}$ et de A, que A refuse $\langle \langle B \rangle, \langle B \rangle \rangle$. Mais si A refuse ce mot, B est construit pour accepter son entrée $\langle B \rangle$. Contradiction.

□

8.3.4 Problèmes semi-décidables

Le problème $\mathrm{HALTING} - \mathrm{PROBLEM}$ est toutefois semi-décidable dans le sens suivant :

Définition 8.10 (Langage semi-décidable) *Un langage* $L \subset M^*$ *est dit* semi-décidable *s'il correspond à l'ensemble des mots acceptés par une machine de Turing* M.

Corollaire 8.1 *Le langage universel* $\mathrm{HALTING} - \mathrm{PROBLEM}$ *est semi-décidable.*

Démonstration : Pour savoir si on doit accepter une entrée qui correspond au codage $\langle M \rangle$ d'une machine de Turing M et au mot w, il suffit de simuler la machine de Turing M sur l'entrée w. On arrête la simulation et on accepte si l'on détecte dans cette simulation que la machine de Turing M atteint un état accepteur. Sinon, on simule M pour toujours. □

Un langage semi-décidable est aussi dit *récursivement énumérable*.

On note RE la classe des langages et des problèmes semi-décidables : voir la figure 8.2.

Corollaire 8.2 $\mathrm{R} \subsetneq \mathrm{RE}$.

Démonstration : L'inclusion est par définition. Puisque $\mathrm{HALTING} - \mathrm{PROBLEM}$ est dans RE et n'est pas dans R, l'inclusion est stricte. □

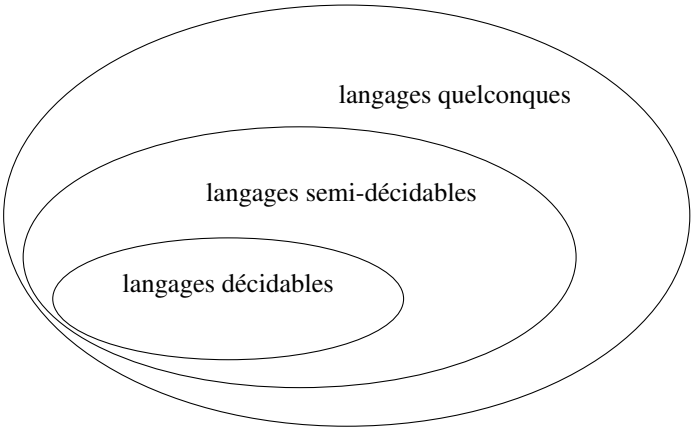

FIGURE 8.2 – Inclusions entre classes de langages.

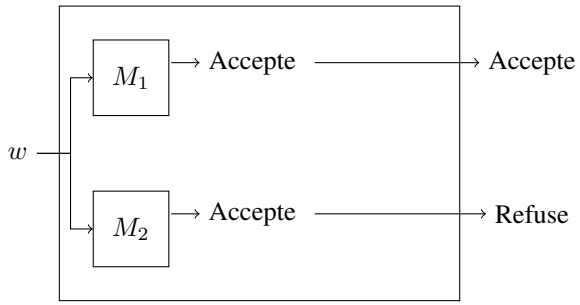

FIGURE 8.3 – Illustration de la preuve du théorème 8.4.

8.3.5 Un problème qui n'est pas semi-décidable

Commençons par établir le résultat fondamental suivant :

Théorème 8.4 *Un langage est décidable si et seulement s'il est semi-décidable et son complémentaire aussi.*

Remarque 8.6 *Ce résultat justifie la terminologie de* semi-décidable*, puisqu'un langage qui est semi-décidable et dont le complémentaire l'est aussi est décidable.*

Démonstration : sens $\Leftarrow$. Supposons que L soit semi-décidable et son complémentaire aussi. Il existe une machine de Turing M_1 qui termine en acceptant sur L, et une machine de Turing M_2 qui termine en acceptant sur son complémentaire. On construit une machine de Turing M qui, sur une entrée w, simule en parallèle [1] M_1 et

1. Une alternative est de considérer que M est une machine de Turing non-déterministe qui simule de façon non-déterministe soit M_1 soit M_2.

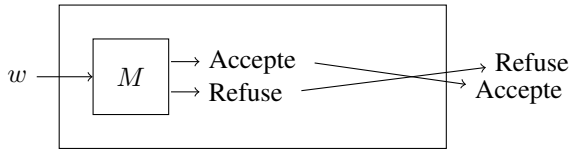

FIGURE 8.4 – Construction d'une machine de Turing acceptant le complément d'un langage décidable.

M_2, (c'est-à-dire que l'on simule t étapes de M_1 sur w, puis on simule t étapes de M_2 sur w, pour $t = 1, 2, \cdots$,) jusqu'à ce que l'une des deux termine : voir la figure 8.3. Si M_1 termine, la machine de Turing M accepte. Si c'est M_2, la machine M refuse. La machine de Turing M que l'on vient de décrire décide L.

sens $\Rightarrow$. Par définition, un langage décidable est semi-décidable. En inversant dans la machine de Turing l'état d'acceptation et de refus, son complémentaire est aussi décidable (voir la figure 8.4) et donc aussi semi-décidable. □

On considère alors le complémentaire du problème $\mathrm{HALTING-PROBLEM}$, que l'on va noter $\overline{\mathrm{HALTING-PROBLEM}}$.

Définition 8.11 ($\overline{\mathrm{HALTING-PROBLEM}}$)
***Donnée**: Le codage $\langle M \rangle$ d'une machine de Turing M et un mot w.*
***Réponse**: Décider si la machine M n'accepte pas le mot w.*

Corollaire 8.3 *Le problème $\overline{\mathrm{HALTING-PROBLEM}}$ n'est pas semi-décidable.*

Démonstration : Sinon, par le théorème précédent, le problème de décision $\mathrm{HALTING-PROBLEM}$ serait décidable. □

8.3.6 Sur la terminologie utilisée

Un langage décidable est aussi appelé un langage *récursif* : la terminologie de *récursif* fait référence à la notion de fonction récursive : voir par exemple le cours [Dowek, 2008] qui présente la calculabilité par l'intermédiaire des fonctions récursives.

La notion de *énumérable* dans *récursivement énumérable* s'explique par le résultat suivant.

Théorème 8.5 *Un langage $L \subset M^*$ est récursivement énumérable si et seulement si l'on peut produire une machine de Turing qui affiche un à un (énumère) tous les mots du langage L.*

Démonstration : sens $\Rightarrow$. Supposons que L soit récursivement énumérable. Il existe une machine de Turing A qui termine en acceptant sur les mots de L.

L'ensemble des couples (t, w), où t est un entier, w est un mot, est dénombrable. On peut se convaincre assez facilement qu'il est en fait effectivement dénombrable : on peut construire une machine de Turing qui produit le codage $\langle t, w \rangle$ de tous les couples (t, w). Par exemple, on considère une boucle qui pour $t = 1, 2, \cdots$ jusqu'à l'infini,

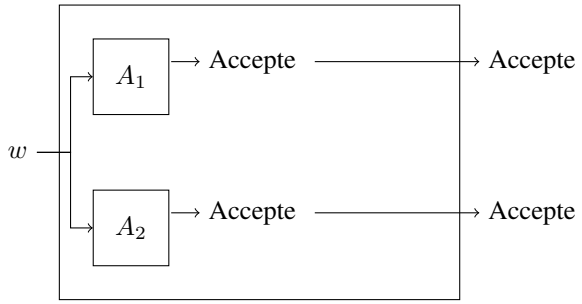

FIGURE 8.5 – Construction d'une machine de Turing acceptant $L_1 \cup L_2$.

considère tous les mots w de longueur inférieure ou égale à t, et produit pour chacun le couple $\langle t, w \rangle$.

Considérons une machine de Turing B qui en plus, pour chaque couple produit (t, w), simule en outre t étapes de la machine A. Si la machine A termine et accepte en exactement t étapes, B affiche alors le mot w. Sinon B n'affiche rien pour ce couple.

Un mot du langage L, est accepté par A en un certain temps t. Il sera alors écrit par B lorsque celui-ci considérera le couple (t, w). Clairement, tout mot w affiché par B est accepté par A, et donc est un mot de L.

sens $\Leftarrow$. Réciproquement, si l'on a une machine de Turing B qui énumère tous les mots du langage L, alors on peut construire une machine de Turing A, qui étant donné un mot w, simule B, et à chaque fois que B produit un mot compare ce mot au mot w. S'ils sont égaux, alors A s'arrête et accepte. Sinon, A continue à jamais.

Par construction, sur une entrée w, A termine et accepte si w se trouve parmi les mots énumérés par B, c'est-à-dire si $w \in L$. Si w ne se trouve pas parmi ces mots, par hypothèse, $w \notin L$, et donc par construction, A n'acceptera pas w. □

8.3.7 Propriétés de clôture

Théorème 8.6 *L'ensemble des langages semi-décidables est clos par union et par intersection : autrement dit, si L_1 et L_2 sont semi-décidables, alors $L_1 \cup L_2$ et $L_1 \cap L_2$ le sont.*

Démonstration : Supposons que L_1 corresponde à $L(A_1)$ pour une machine de Turing A_1 et L_2 à $L(A_2)$ pour une machine de Turing A_2. Alors $L_1 \cup L_2$ correspond à $L(A)$ pour la machine de Turing A qui simule en parallèle A_1 et A_2 et qui termine et accepte dès que l'une des deux machines de Turing A_1 et A_2 termine et accepte : voir la figure 8.5.

$L_1 \cap L_2$ correspond à $L(A)$ pour la machine de Turing A qui simule en parallèle A_1 et A_2 et qui termine dès que les deux machines de Turing A_1 et A_2 terminent et acceptent.

□

Théorème 8.7 *L'ensemble des langages décidables est clos par union, intersection, et complément : autrement dit, si L_1 et L_2 sont décidables, alors $L_1 \cup L_2$, $L_1 \cap L_2$, et L_1^c le sont.*

Démonstration : On a déjà utilisé la clôture par complémentaire : en inversant dans la machine de Turing l'état d'acceptation et de refus, le complémentaire d'un langage décidable est aussi décidable (voir la figure 8.4).

Il reste à montrer qu'avec les hypothèses, les langages $L_1 \cup L_2$ et $L_1 \cap L_2$ sont décidables. Mais cela est clair par le théorème précédent et le fait qu'un ensemble est décidable si et seulement si il est semi-décidable et son complémentaire aussi, en utilisant les lois de Morgan (le complémentaire d'une union est l'intersection des complémentaires, et inversement) et la stabilité par complémentaire des langages décidables. □

8.4 Autres problèmes indécidables

Une fois que nous avons obtenu un premier problème indécidable, nous allons en obtenir d'autres.

8.4.1 Réductions

Nous connaissons deux langages indécidables, $\mathrm{HALTING-PROBLEM}$ et son complémentaire. Notre but est maintenant d'en obtenir d'autres, et de savoir comparer les problèmes. Nous introduisons pour cela la notion de *réduction*.

Tout d'abord, nous pouvons généraliser la notion de *calculable* aux fonctions et pas seulement aux langages et problèmes de décision.

Définition 8.12 (Fonction calculable) *Soient Σ et Σ' deux alphabets. Une fonction $f : \Sigma^* \to \Sigma'^*$ (totale) est* calculable *s'il existe une machine de Turing A, qui travaille sur l'alphabet $\Sigma \cup \Sigma'$, telle que pour tout mot w, A avec l'entrée w, termine et accepte, avec $f(w)$ écrit sur son ruban au moment où elle termine.*

On peut se convaincre facilement que la composée de deux fonctions calculables est calculable.

Cela nous permet d'introduire une notion de réduction entre problèmes : l'idée est que si A se réduit à B, alors le problème A est plus facile que le problème B, ou si l'on préfère, le problème B est plus difficile que le problème A : voir la figure 8.6 et la figure 8.7.

Définition 8.13 (Réduction) *Soient A et B deux problèmes d'alphabet respectifs M_A et M_B. Une* réduction de A vers B *est une fonction $f : M_A^* \to M_B^*$ calculable telle que $w \in A$ ssi $f(w) \in B$. On note $A \leq_m B$ lorsque A se réduit à B.*

Cela se comporte comme on le souhaite : un problème est aussi facile (et difficile) que lui-même, et la relation "être plus facile que" est transitive. En d'autres termes :

Théorème 8.8 *$\leq_m$ est un préordre :*

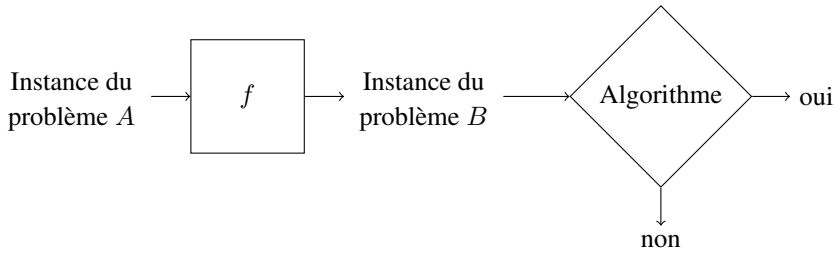

FIGURE 8.6 – Réduction du problème A vers le problème B. Si l'on peut résoudre le problème B, alors on peut résoudre le problème A. Le problème A est donc plus facile que le problème B, noté $A \leq_m B$.

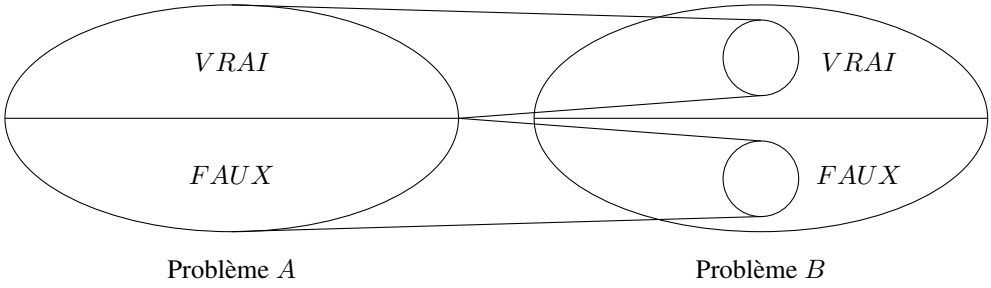

FIGURE 8.7 – Les réductions transforment des instances positives en instances positives, et négatives en négatives.

1. $L \leq_m L$;
2. $L_1 \leq_m L_2$, $L_2 \leq_m L_3$ *impliquent* $L_1 \leq_m L_3$.

Démonstration : Considérer la fonction identité comme fonction f pour le premier point.

Pour le second point, supposons $L_1 \leq_m L_2$ via la réduction f, et $L_2 \leq_m L_3$ via la réduction g. On a $x \in L_1$ ssi $g(f(x)) \in L_2$. Il suffit alors de voir que $g \circ f$, en temps que composée de deux fonctions calculables est calculable. □

Remarque 8.7 *Il ne s'agit pas d'un ordre, puisque $L_1 \leq_m L_2$, $L_2 \leq_m L_1$ n'implique pas $L_1 = L_2$. Il est en fait naturel d'introduire le concept suivant : deux problèmes L_1 et L_2 sont équivalents, noté $L_1 \equiv L_2$, si $L_1 \leq_m L_2$ et si $L_2 \leq_m L_1$. On a alors $L_1 \leq_m L_2$, $L_2 \leq_m L_1$ impliquent $L_1 \equiv L_2$.*

Intuitivement, si un problème est plus facile qu'un problème décidable, alors il est décidable. Formellement :

Proposition 8.2 (Réduction) *Si $A \leq_m B$, et si B est décidable alors A est décidable*
.

Démonstration : A est décidé par la machine de Turing qui, sur une entrée w, calcule $f(w)$, puis simule la machine de Turing qui décide B sur l'entrée $f(w)$. Puisqu'on a $w \in A$ si et seulement si $f(w) \in B$, la machine de Turing est correcte. □

Proposition 8.3 (Réduction) *Si $A \leq_m B$, et si A est indécidable, alors B est indécidable.*

Démonstration : Il s'agit de la contraposée de la proposition précédente. □

8.4.2 Quelques autres problèmes indécidables

Cette idée va nous permettre d'obtenir immédiatement la preuve de l'indécidabilité de plein d'autres problèmes.

Par exemple, le fait qu'il n'est pas possible de déterminer algorithmiquement si une machine de Turing accepte au moins une entrée :

Définition 8.14 (Problème $L_\emptyset$)
***Donnée**: Le codage $\langle M \rangle$ d'une machine de Turing M.*
***Réponse**: Décider si $L(M) \neq \emptyset$.*

Proposition 8.4 *Le problème Problème $L_\emptyset$ est indécidable.*

Démonstration : On construit une réduction de $\mathrm{HALTING} - \mathrm{PROBLEM}$ vers $L_\emptyset$: pour toute paire $\langle \langle A \rangle, w \rangle$, on considère la machine de Turing A_w définie de la manière suivante (voir la figure 8.8) :

- A_w prend en entrée un mot u ;
- A_w simule A sur w ;
- Si A accepte w, alors A_w accepte.

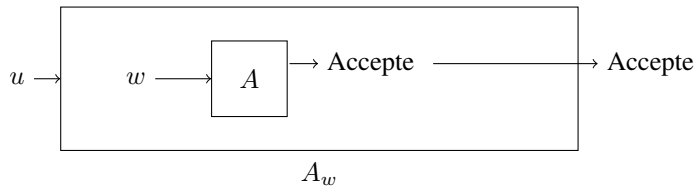

FIGURE 8.8 – Illustration de la machine de Turing utilisée dans la preuve de la proposition 8.4.

La fonction f qui à $\langle\langle A\rangle, w\rangle$ associe $\langle A_w\rangle$ est bien calculable. De plus on a $\langle\langle A\rangle, w\rangle \in \text{HALTING} - \text{PROBLEM}$ si et seulement si $L(A_w) \neq \emptyset$, c'est-à-dire $\langle A_w\rangle \in L_\emptyset$: en effet, A_w accepte soit tous les mots (et donc le langage correspondant n'est pas vide) si A accepte w, soit n'accepte aucun mot (et donc le langage correspondant est vide) sinon. □

Définition 8.15 (Problème $L_\neq$)
***Donnée**: Le codage $\langle A\rangle$ d'une machine de Turing A et le codage $\langle A'\rangle$ d'une machine de Turing A'.*
***Réponse**: Déterminer si $L(A) \neq L(A')$.*

Proposition 8.5 *Le problème Problème $L_\neq$ est indécidable.*

Démonstration : On construit une réduction de $L_\emptyset$ vers $L_\neq$. On considère une machine de Turing fixe B qui accepte le langage vide : prendre par exemple une machine de Turing B qui rentre immédiatement dans une boucle sans fin. La fonction f qui à $\langle A\rangle$ associe la paire $\langle A, B\rangle$ est bien calculable. De plus on a $\langle A\rangle \in L_\emptyset$ si et seulement si $L(A) \neq \emptyset$ si et seulement si $\langle A, B\rangle \in L_\neq$. □

8.4.3 Théorème de Rice

Les deux résultats précédents peuvent être vus comme les conséquences d'un résultat très général qui affirme que toute propriété non triviale des algorithmes est indécidable.

Théorème 8.9 (Théorème de Rice) *Toute propriété non triviale des langages semi-décidables est indécidable.*

Autrement dit, soit une propriété P des langages semi-décidables non triviale, c'est-à-dire telle qu'il y a au moins une machine de Turing M telle que $L(M)$ satisfait P et une machine de Turing M telle que $L(M)$ ne satisfait pas P.

Alors le problème de décision L_P :
***Donnée**: Le codage $\langle M\rangle$ d'une machine de Turing M ;*
***Réponse**: Décider si $L(M)$ vérifie la propriété P ;*
est indécidable.

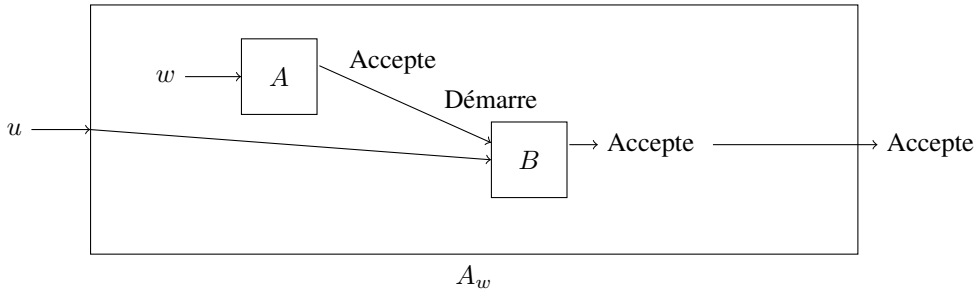

FIGURE 8.9 – Illustration de la machine de Turing utilisée dans la preuve du théorème de Rice.

Remarque 8.8 *Remarquons que si une propriété P est triviale, L_P est décidable trivialement : construire une machine de Turing qui ne lit même pas son entrée et qui accepte (respectivement : refuse).*

Démonstration : Il nous faut démontrer que le problème de décision L_P est indécidable.

Quitte à remplacer P par sa négation, on peut supposer que le langage vide ne vérifie pas la propriété P (prouver l'indécidabilité de L_p est équivalent à prouver l'indécidabilité de son complémentaire). Puisque P est non triviale, il existe au moins une machine de Turing B avec $L(B)$ qui vérifie P.

On construit une réduction de $\mathrm{HALTING} - \mathrm{PROBLEM}$ vers le langage L_P. Étant donnée une paire $\langle\langle A\rangle, w\rangle$, on considère la machine de Turing A_w définie de la façon suivante (voir la figure 8.9) :

- A_w prend en entrée un mot u ;
- Sur le mot u, A_w simule A sur le mot w ;
- Si A accepte w, alors A_w simule B sur le mot u : A_w accepte si seulement si B accepte u.

Autrement dit, A_w accepte, si et seulement si A accepte w et si B accepte u. Si w est accepté par A, alors $L(A_w)$ vaut $L(B)$, et donc vérifie la propriété P. Si w n'est pas accepté par A, alors $L(A_w) = \emptyset$, et donc ne vérifie pas la propriété P.

La fonction f qui à $\langle\langle A\rangle, w\rangle$ associe $\langle A_w\rangle$ est bien calculable. □

Exercice 8.2. *Montrer que l'ensemble des codages des machines de Turing qui acceptent tous les mots qui sont des palindromes (en acceptant possiblement d'autres entrées) est indécidable.*

8.4.4 Le drame de la vérification

Autrement dit :

Corollaire 8.4 *Il n'est pas possible de construire un algorithme qui prendrait en entrée un programme, et sa spécification, et qui déterminerait si le programme satisfait*

sa spécification.

Et cela, en fait, même si l'on fixe la spécification à une propriété P (dès que la propriété P n'est pas triviale), par le théorème de Rice.

Par les discussions du chapitre précédent, cela s'avère vrai même pour des systèmes extrêmements rudimentaires. Par exemple :

Corollaire 8.5 *Il n'est pas possible de construire un algorithme qui prendrait en entrée la description d'un système, et sa spécification, et qui déterminerait si le système satisfait sa spécification.*

Et cela, en fait, même si l'on fixe la spécification à une propriété P (dès que la propriété P n'est pas triviale), et même pour des systèmes aussi simples que les machines à 2-compteurs, par le théorème de Rice, et les résultats de simulation du chapitre précédent.

8.4.5 Notion de complétude

Notons que l'on peut aussi introduire une notion de complétude.

Définition 8.16 (RE**-complétude)** *Un problème A est dit* RE*-complet, si :*

1. *il est récursivement énumérable ;*
2. *tout autre problème récursivement énumérable B est tel que $B \leq_m A$.*

Autrement dit, un problème RE-complet est maximal pour $\leq_m$ parmi les problèmes de la classe RE.

Théorème 8.10 *Le problème* $\mathrm{HALTING} - \mathrm{PROBLEM}$ *est* RE*-complet.*

Démonstration : $\mathrm{HALTING} - \mathrm{PROBLEM}$ est semi-décidable. Maintenant, soit L un langage semi-décidable. Par définition, il existe une machine de Turing A qui accepte L. Considérons la fonction f qui à w associe le mot $\langle\langle A\rangle, w\rangle$. On a $w \in L$ si et seulement si $f(w) \in \mathrm{HALTING} - \mathrm{PROBLEM}$, et donc $L \leq_m \mathrm{HALTING} - \mathrm{PROBLEM}$. □

8.5 Problèmes indécidables naturels

On peut objecter que les problèmes précédents, relatifs aux algorithmes sont "artificiels", dans le sens où ils parlent de propriétés d'algorithmes, les algorithmes ayant eux-même été définis par la théorie de la calculabilité.

Il est difficile de définir formellement ce qu'est un problème *naturel*, mais on peut au moins dire qu'un problème qui a été discuté avant l'invention de la théorie de la calculabilité est (plus) naturel.

8.5.1 Le dixième problème de Hilbert

C'est clairement le cas du dixième problème identifié par Hilbert parmi les problèmes intéressants pour le 20ème siècle en 1900 : peut-on déterminer si une équation polynomiale à coefficients entiers possède une solution entière.

Définition 8.17 (Dixième problème de Hilbert)
***Donnée**: Un polynôme $P \in \mathbb{N}[X_1, \cdots, X_n]$ à coefficients entiers.*
***Réponse**: Décider s'il possède une racine entière.*

Théorème 8.11 *Le problème Dixième problème de Hilbert est indécidable.*

La preuve de ce résultat, due à Matiyasevich [Matiyasevich, 1970] est hors de l'ambition de ce document.

8.5.2 Le problème de la correspondance de Post

La preuve de l'indécidabilité du problème de la correspondance de Post est plus facile, même si nous ne la donnerons pas ici. On peut considérer ce problème comme "naturel", dans le sens où il ne fait pas référence directement à la notion d'algorithme, ou aux machines de Turing :

Définition 8.18 (Problème de la correspondance de Post**)**
***Donnée**: Une suite $(u_1, v_1), \cdots, (u_n, v_n)$ de paires de mots sur l'alphabet Σ.*
***Réponse**: Décider cette suite admet une* solution*, c'est-à-dire une suite d'indices $i_1, i_2, \cdots, i_m$ de $\{1, 2, \cdots, n\}$ telle que*

$$u_{i_1} u_{i_2} \cdots u_{i_m} = v_{i_1} v_{i_2} \cdots v_{i_m}.$$

Théorème 8.12 *Le problème* Problème de la correspondance de Post *est indécidable.*

8.5.3 Décidabilité/Indécidabilité de théories logiques

Nous avons déjà présenté dans le chapitre 6Modèles. Complétudechapter.6, les axiomes de l'arithmétique de Robinson et les axiomes de Peano : on s'attend à ce que ces axiomes soient vérifiés sur les entiers, c'est-à-dire dans *le modèle standard des entiers* où l'ensemble de base est les entiers, et où l'on interprète $+$ par l'addition, $*$ par la multiplication, et $s(x)$ par $x + 1$.

Étant donnée une formule close sur une signature contenant les symboles, F est soit vraie soit fausse sur les entiers (c'est-à-dire dans le modèle standard des entiers). Appelons *théorie de l'arithmétique*, l'ensemble $Th(\mathbb{N})$ des formules closes F qui sont vraies sur les entiers.

Les constructions du chapitre suivant prouvent le résultat suivant :

Théorème 8.13 *$Th(\mathbb{N})$ n'est pas décidable.*

Démonstration : Nous prouvons dans le chapitre qui suit que $Th(\mathbb{N})$ n'est pas récursivement énumérable. Il suffit d'observer qu'un ensemble décidable est récursivement énumérable, pour obtenir une contradiction à supposer $Th(\mathbb{N})$ décidable. □

On peut prouver (ce que l'on ne fera pas) que si l'on considère des formules sans utiliser le symbole de multiplication, alors la théorie correspondante est décidable.

Théorème 8.14 *On peut décider si une formule close F sur la signature $(\mathbf{0}, s, +, =)$ (i.e. celle de Peano sans la multiplication) est satisfaite sur les entiers standards.*

On obtient aussi par ailleurs les résultats suivants :

Théorème 8.15 *Soit F une formule close sur la signature des axiomes de Peano. Le problème de décision consistant à déterminer si F peut se prouver à partir des axiomes de Peano est indécidable.*

Démonstration : Étant donné $\langle\langle M\rangle, w\rangle$, où M est une machine et w un mot, nous montrons dans le chapitre qui suit comment produire une formule close γ sur la signature de l'arithmétique telle que

$$\langle\langle M\rangle, w\rangle \in \overline{\mathrm{HALTING-PROBLEM}} \Leftrightarrow \gamma \in Th(\mathbb{N}),$$

où $\overline{\mathrm{HALTING-PROBLEM}}$ est le complémentaire du problème de l'arrêt des machines de Turing.

Or, en fait, on peut se convaincre que le raisonnement que l'on utilise pour cela peut se formaliser avec Peano et se déduire des axiomes de Peano.

On a donc en fait $\langle\langle M\rangle, w\rangle \in \overline{\mathrm{HALTING-PROBLEM}}$ si et seulement si γ se prouve à partir des axiomes de Peano.

Cela donne une réduction du complémentaire du problème de l'arrêt des machines de Turing vers notre problème : la transformation qui à $\langle\langle M\rangle, w\rangle$ associe γ se calcule bien facilement. Notre problème est donc indécidable, puisque le premier l'est. □

On peut prouver :

Théorème 8.16 *On peut décider si une formule close F sur la signature $(\mathbf{0}, s, +, =)$ (i.e. celle de Peano sans la multiplication) est prouvable à partir des axiomes de Peano.*

8.6 Théorèmes du point fixe

Les résultats de cette section sont très subtils, mais extrêmement puissants.

Commençons par une version simple, qui nous aidera à comprendre les preuves.

Proposition 8.6 *Il existe une machine de Turing A^* qui écrit son propre algorithme : elle produit en sortie $\langle A^*\rangle$.*

Autrement dit, il est possible d'écrire un programme qui affiche son propre code.

C'est vrai dans tout langage de programmation qui est équivalent aux machines de Turing :

En shell *UNIX* par exemple, le programme suivant

```
x='y='echo . | tr . "\47" '; echo "x=$y$x$y;$x"'; y='echo . | tr .
 "\47"'; echo "x=$y$x$y;$x"
```

produit

```
x='y='echo . | tr . "\47" '; echo "x=$y$x$y;$x"'; y='echo . | tr .
 "\47"'; echo "x=$y$x$y;$x"
```

qui est une commande, qui exécutée, affiche son propre code.

On appelle parfois de tels programme des *quines*, en l'honneur du philosophe Willard van Orman Quine, qui a discuté l'existence de programmes autoreproducteurs.

Démonstration : On considère des machines de Turing qui terminent sur toute entrée. Pour deux telles machines A et A', on note AA' la machine de Turing qui est obtenue en composant de façon séquentielle A et A'. Formellement, AA' est la machine de Turing qui effectue d'abord le programme de A, et puis lorsque A temine avec sur son ruban w, effectue le programme de A' sur l'entrée w.

On construit les machines suivantes :

1. Étant donné un mot w, la machine de Turing $Print_w$ termine avec le résultat w ;
2. Pour une entrée w de la forme $w = \langle X \rangle$, où X est une machine de Turing, la machine de Turing B produit en sortie le codage de la machine de Turing $Print_w X$, c'est-à-dire le codage de la machine de Turing obtenue en composant $Print_w$ et X.

On considère alors la machine de Turing A^* donnée par $Print_{\langle B \rangle} B$, c'est-à-dire la composition séquentielle des machines $Print_{\langle B \rangle}$ et B.

Déroulons le résultat de cette machine : la machine de Turing $Print_{\langle B \rangle}$ produit en sortie $\langle B \rangle$. La composition par B produit alors le codage de $Print_{\langle B \rangle} \langle B \rangle$, qui est bien le codage de la machine de Turing A^*. □

Le théorème de récursion permet des autoréférences dans un langage de programmation. Sa démonstration consiste à étendre l'idée derrière la preuve du résultat précédent.

Théorème 8.17 (Théorème de récursion) *Soit $t : M^* \times M^* \to M^*$ une fonction calculable. Alors il existe une machine de Turing R qui calcule une fonction $r : M^* \to M^*$ telle que pour tout mot w*

$$r(w) = t(\langle R \rangle, w).$$

Son énoncé peut paraître technique, mais son utilisation reste assez simple. Pour obtenir une machine de Turing qui obtient sa propre description et qui l'utilise pour calculer, on a besoin simplement d'une machine de Turing T qui calcule une fonction t comme dans l'énoncé, qui prend une entrée supplémentaire qui contient la description de la machine de Turing. Alors le théorème de récursion produit une nouvelle machine R qui opère comme T mais avec la description de $\langle R \rangle$ gravée dans son code.

Démonstration : Il s'agit de la même idée que précédemment. Soit T une machine de Turing calculant la fonction t : T prend en entrée une paire $\langle u, w \rangle$ et produit en sortie un mot $t(u, w)$.

On considère les machines suivantes :

1. Étant donné un mot w, la machine de Turing $Print_w$ prend en entrée un mot u et termine avec le résultat $\langle w, u\rangle$;
2. Pour une entrée w' de la forme $\langle\langle X\rangle, w\rangle$, la machine de Turing B :
   (a) calcule $\langle\langle Print_{\langle X\rangle}X\rangle, w\rangle$, où $Print_{\langle X\rangle}X$ désigne la machine de Turing qui compose $Print_{\langle X\rangle}$ avec X ;
   (b) puis passe le contrôle à la machine de Turing T.

On considère alors la machine de Turing R donnée par $Print_{\langle B\rangle}B$, c'est-à-dire la machine de Turing obtenue en composant $Print_{\langle B\rangle}$ avec B.

Déroulons le résultat $r(w)$ de cette machine R sur une entrée w : sur une entrée w, la machine de Turing $Print_{\langle B\rangle}$ produit en sortie $\langle\langle B\rangle, w\rangle$. La composition par B produit alors le codage de $\langle\langle Print_{\langle B\rangle}B\rangle, w\rangle$, et passe le contrôle à T. Ce dernier produit alors $t(\langle Print_{\langle B\rangle}B\rangle, w) = t(\langle R\rangle, w) = r(w)$. □

On obtient alors :

Théorème 8.18 (Théorème du point fixe de Kleene) *Soit une fonction calculable qui à chaque mot $\langle A\rangle$ codant une machine de Turing associe un mot $\langle A'\rangle$ codant une machine de Turing. Notons $A' = f(A)$.*

Alors il existe une machine de Turing A^* tel que $L(A^*) = L(f(A^*))$.

Démonstration : Considérons une fonction $t : M^* \times M^* \to M^*$ telle que $t(\langle A\rangle, x)$ soit le résultat de la simulation de la machine de Turing $f(A)$ sur l'entrée x. Par le théorème précédent, il existe une machine de Turing R qui calcule une fonction r telle que $r(w) = t(\langle R\rangle, w)$. Par construction $A^* = R$ et $f(A^*) = f(R)$ ont donc même valeur sur w pour tout w. □

Remarque 8.9 *On peut interpréter les résultat précédents en lien avec les virus informatiques. En effet, un virus est un programme qui vise à se diffuser, c'est-à-dire à s'autoreproduire, sans être détecté. Le principe de la preuve du théorème de récursion est un moyen de s'autoreproduire, en dupliquant son code.*

8.7 Plusieurs remarques

8.7.1 Calculer sur d'autres domaines

Nous avons introduit la notion d'ensemble décidable, récursivement énumérable, etc. pour les sous-ensembles de Σ^*, et la notion de fonction (totale) calculable $f : \Sigma^* \to \Sigma^*$.

On peut vouloir travailler sur d'autres domaines que les mots sur un alphabet donné, par exemple sur les entiers. Dans ce cas, il suffit de considérer que l'on code un entier par son écriture en, disons, base 2 pour se ramener au cas d'un mot sur l'alphabet $\{0, 1\}$. On peut aussi coder un entier par exemple en unaire, i.e. n par a^n pour une lettre a sur un alphabet Σ avec $a \in \Sigma$.

Dans le cas général, pour travailler sur un domaine E, on fixe un codage des éléments de E sur un alphabet Σ : on dit par exemple qu'un sous-ensemble $S \subset E$ est

récursivement énumérable (respectivement décidable), si le sous-ensemble des codages des mots de E est récursivement énumérable (resp. décidable).

De même une fonction (totale) $f : E \to F$ est dit calculable si la fonction qui passe du codage de $e \in E$ au codage de $f \in F$ est calculable.

Exemple 8.5 *On peut coder $\vec{n} = (n_1, \ldots, n_k) \in \mathbb{N}^k$ par*

$$\langle \vec{n} \rangle = a^{n_1+1} b a^{n_2+1} b \cdots a^{n_k+1}$$

sur l'alphabet $\Sigma = \{a, b\}$. Une fonction $f : \mathbb{N}^k \to \mathbb{N}$ est dite calculable si elle est calculable sur ce codage.

Les notions obtenues de fonction calculable, ensemble décidable, semi-décidable, etc... ne dépendent pas du codage, pour les codages usuels (en fait techniquement dès que l'on passer d'un codage à un autre de façon calculable).

8.7.2 Vision algébrique de la calculabilité

Les notions de la calculabilité sont parfois introduites de façon algébrique, en parlant de fonctions sur les entiers.

En particulier, on peut introduire la notion de fonction partielle calculable, qui étend la notion de fonction calculable introduite au cas des fonctions non-totales.

Définition 8.19 (Fonction partielle calculable) *Soit $f : E \to F$ une fonction, possiblement partielle.*

La fonction $f : E \to F$ est calculable *s'il existe une machine de Turing A telle que pour tout mot codage w d'un élement $e \in E$ dans le domaine de f, A avec l'entrée w, termine et accepte, avec le codage de $f(e)$ écrit sur son ruban au moment où elle termine.*

Bien entendu, cette notion correspond à la notion précédente pour le cas d'une fonction f totale.

On peut caractériser de façon purement algébrique la notion de fonction calculable :

Définition 8.20 (Fonction récursive) *Une fonction (possiblement partielle) $f : \mathbb{N}^n \to \mathbb{N}$ est* récursive *si elle est soit la constante 0, soit l'une des fonctions :*

- $\mathsf{Zero} : x \mapsto 0$ *la fonction 0 ;*
- $\mathsf{Succ} : x \mapsto x + 1$ *la fonction successeur ;*
- $\mathsf{Proj}_n^i : (x_1, \ldots, x_n) \mapsto x_i$ *les fonctions de projection, pour $1 \leq i \leq n$;*
- $\mathsf{Comp}_m(g, h_1, \ldots, h_m) : (x_1, \ldots, x_n) \mapsto g(h_1(x_1, \ldots, x_n), \ldots, h_m(x_1, \ldots, x_n))$ *la composition des fonctions récursives $g, h_1, \ldots, h_m$;*
- $\mathsf{Rec}(g, h)$ *la fonction définie par récurrence comme*

$$\begin{cases} f(0, x_2, \ldots, x_n) = g(x_2, \ldots, x_n), \\ f(x_1 + 1, x_2, \ldots, x_n) = h(f(x_1, \ldots, x_n), x_1, \ldots, x_n), \end{cases}$$

où g et h sont récursives.

– $\mathsf{Min}(g)$ *la fonction qui à* $(x_2, \ldots, x_n)$ *associe le plus petit* $y \in \mathbb{N}$ *tel que*

$$g(y, x_2, \ldots, x_n) = 1$$

s'il en existe (et qui n'est pas définie sinon) où g *est récursive.*

Une fonction primitive récursive *est une fonction qui se définit sans utiliser le schéma* Min.

On peut prouver le résultat suivant :

Théorème 8.19 *Une fonction* $f : \mathbb{N}^n \to \mathbb{N}$ *est récursive si et seulement si elle est calculable par une machine de Turing.*

La notion d'ensemble décidable ou semi-décidable peut se définir aussi de façon algébrique :

Théorème 8.20 *Un sous-ensemble* $S \subset \mathbb{N}$ *est décidable si la fonction charactéristique de* S*, i.e. la fonction (totale)* $\chi : n \mapsto \begin{cases} 1 & \text{si } n \in S \\ 0 & \text{si } n \notin S \end{cases}$ *est calculable*

Théorème 8.21 *Un sous-ensemble* $S \subset \mathbb{N}$ *est semi-décidable si la fonction (partielle)* $n \mapsto \begin{cases} 1 & \text{si } n \in S \\ \text{indéfini} & \text{si } n \notin S \end{cases}$ *est calculable.*

Exercice 8.3. *Prouver ces théorèmes.*

***Exercice 8.4.** *[Problème de l'arrêt généralisé] Soit* A *un sous-ensemble décidable de l'ensemble des codages de machines de Turing, tel que toutes les machines de* A *terminent toujours.*

Alors A *est incomplet : il existe une fonction (unaire)* $f : \mathbb{N} \to \mathbb{N}$ *calculable totale qui n'est représentée par aucune machine de Turing de* A*.*

Expliquer pourquoi ce résultat implique le problème de l'arrêt.

8.8 Exercices

Exercice 8.5. *Soit* $E \subset \mathbb{N}$ *un ensemble récursivement énumérable, énuméré par une fonction calculable* f *strictement croissante. Montrer que* E *est décidable.*

Exercice 8.6. *En déduire que tout sous-ensemble récursivement énumérable infini de* $\mathbb{N}$ *contient un ensemble décidable infini.*

Exercice 8.7. *Soit* $E \subset \mathbb{N}$ *un ensemble décidable. Montrer qu'il est énuméré par une fonction calculable* f *strictement croissante.*

Exercice 8.8. *Soit* $A \subset \mathbb{N}^2$ *un ensemble décidable de couples d'entiers.*

On note $\exists A$ *pour la (première) projection de* A*, à savoir le sous-ensemble de* $\mathbb{N}$ *défini par*

$$\exists A = \{x | \exists y \in \mathbb{N} \text{ tel que } (x, y) \in A\}.$$

1. *Montrer que la projection d'un ensemble décidable est récursivement énumérable.*
2. *Montrer que tout ensemble récursivement énumérable est la projection d'un ensemble décidable.*

Exercice 8.9. *Un nombre réel a est dit* récursif *s'il existe des fonctions calculables F et G de $\mathbb{N}$ dans $\mathbb{N}$ telles que pour tout $n > 0$ on ait $G(n) > 0$ et*

$$||a| - \frac{F(n)}{G(n)}| \leq \frac{1}{n}.$$

1. *Montrer que tout nombre rationnel est récursif.*
2. *Montrer que $\sqrt{2}$, π, e sont récursifs.*
3. *Montrer que le nombre réel $0 < a < 1$ est récursif si et seulement s'il existe un développement décimal récursif de a, c'est-à-dire une fonction calculable $H : \mathbb{N} \to \mathbb{N}$ telle que pour tout $n > 0$ on ait $0 \leq H(n) \leq 9$ et*

$$|a| = \sum_{n=0}^{\infty} \frac{H(n)}{10^n}.$$

4. *Montrer que l'ensemble des réels récursifs forme un sous corps dénombrable de $\mathbb{R}$, tel que tout polynôme de degré impair possède une racine.*
5. *Donner un exemple de réel non-récursif.*

Exercice 8.10. *Un état "inutile" d'une machine de Turing est un état $q \in Q$ dans lequel on ne rentre sur aucune entrée. Formuler la question de savoir si une machine de Turing possède un état inutile comme un problème de décision, et prouver qu'il est indécidable.*

Exercice 8.11. *Les problèmes suivants, où l'on se donne une machine de Turing A et l'on veut savoir*

1. *si $L(A)$ (le langage accepté par A) contient au moins deux entrées distinctes*
2. *si $L(A)$ accepte aucune entrée*

sont-ils décidables ? semi-décidables ?

8.9 Notes bibliographiques

Lectures conseillées Pour aller plus loin sur les notions évoquées dans ce chapitre, nous suggérons la lecture de [Sipser, 1997] et de [Hopcroft et al., 2001] en anglais, ou de [Wolper, 2001], [Stern, 1994] [Carton, 2008] en français.

Le livre [Sipser, 1997] est un excellent ouvrage très pédagogue.

Bibliographie Ce chapitre contient des résultats standards en calculabilité. Nous nous sommes inspirés ici essentiellement de leur présentation dans les livres [Wolper, 2001], [Carton, 2008], [Jones, 1997], [Kozen, 1997], [Hopcroft et al., 2001], ainsi que dans [Sipser, 1997].

Index

Bibliographie

[Carton, 2008] Carton, O. (2008). Langages formels, calculabilité et complexité.

[Dowek, 2008] Dowek, G. (2008). *Les démonstrations et les algorithmes*. Polycopié du cours de l'Ecole Polytechnique.

[Hopcroft et al., 2001] Hopcroft, J. E., Motwani, R., and Ullman, J. D. (2001). *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2nd edition.

[Jones, 1997] Jones, N. (1997). *Computability and complexity, from a programming perspective*. MIT press.

[Kozen, 1997] Kozen, D. (1997). *Automata and computability*. Springer Verlag.

[Matiyasevich, 1970] Matiyasevich, Y. (1970). Enumerable sets are diophantine. *Doklady Akademii Nauk SSSR*, 191(2) :279–282.

[Sipser, 1997] Sipser, M. (1997). *Introduction to the Theory of Computation*. PWS Publishing Company.

[Stern, 1994] Stern, J. (1994). Fondements mathématiques de l'informatique. *Ediscience International, Paris*.

[Wolper, 2001] Wolper, P. (2001). *Introduction à la calculabilité : cours et exercices corrigés*. Dunod.