UCLouvain

INGI1123
Calculabilité et Complexité

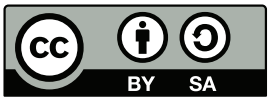
Notes de cours collaboratives

Edited by

Yves Deville

Unversité catholique de Louvain

29 mai 2019



Ce document est un syllabus collaboratif pour le cours *INGI1123 Calculabilité et complexité* donné à l'Université catholique de Louvain par Yves Deville. Ces notes sont en cours d'élaboration et sont le fruit d'un travail collaboratif entre les étudiants de ce cours, avec une supervision de Yves Deville et différents chercheurs et assistants.

Ce document se base sur la *Synthèse de Calculabilité Q6 - LINGI1123*, produite notamment par Floran Hachez, Lionel Nobel, Lena Peschke et François Robinet lorsqu'ils ont suivi ce cours. Cette synthèse a été rédigée sous licence Creative Commons CC-BY-SA et est disponible à l'adresse suivante `https://github.com/Gp2mv3/Syntheses`

Ces notes contiennent certainement des erreurs. Soyez vigilants. Vos contributions sont les bienvenues pour améliorer et corriger ces notes à l'adresse suivante `https://github.com/UCL-INGI/LINGI1123-Calculabilite`

Table des matières

Chapitre 1

Introduction

1.1 Qu'est-ce que la calculabilité

La calculabilité est l'étude des limites de l'informatique. Il faut savoir différencier les limites théoriques des limites pratiques. Pour la calculabilité, on s'occupe des limites théoriques alors que pour la complexité on s'occupe des limites pratiques. La complexité détermine la frontière entre ce qui est faisable et infaisable en pratique. La question principale de la calculabilité est : « quels sont les *problèmes* qui (ne) peuvent (pas) être résolus par un *programme* ? ». De manière informelle, un problème est dit *calculable* s'il existe un programme qui résout ce problème. La caractéristique d'un problème calculable ne donne donc aucune autre information que la preuve de l'existence d'un programme capable de résoudre ce problème. Mais lorsqu'un problème est non calculable, cela nous informe qu'il est inutile d'essayer d'écrire un programme pour résoudre ce problème ; un tel programme n'existe pas !

Le but est donc de tracer des frontières entre les programmes calculables, non calculables et non calculables en pratique.

1.1.1 Exemples de limites

De nombreuses limites existent en informatique. Par exemple, il est impossible de déterminer lorsqu'un programme se termine (cfr. Théorème de Rice 3.9), de déterminer si un programme est écrit sans bugs ou encore d'écrire un programme sachant déterminer si deux programmes sont équivalents.

Mais des limites théoriques existent aussi dans des domaines scientifiques autres que l'informatique.

En physique par exemple, les lois de la thermodynamique établissent le fait que l'on ne peut créer de l'énergie à partir de rien (principe de Lavoisier notamment). Dans un autre registre, on sait qu'il est impossible d'observer, avec la précision que l'on souhaite, en même temps la *position* et la *vitesse* d'un électron.

En mécanique, on sait qu'il est impossible de construire une machine réalisant un mouvement perpétuel.

En mathématique, il est impossible de trouver deux nombres entiers m et n tels que $m^2 = 2n^2$, ou de trouver une fraction $\frac{m}{n}$ exprimant le rapport entre le rayon d'un cercle et sa circonférence, ou encore de diviser un angle en trois parties égales en utilisant uniquement une règle et un compas.

1.2 Notion de problème

Premièrement, on doit parler de la notion de problème. Attention, il ne faut pas confondre un problème avec un programme. Les caractéristiques d'un problème sont :

— un problème est *générique* : il s'applique à un ensemble de données.

— pour chaque donnée particulière, il existe une réponse.

On représente un problème dans le cours par une fonction (noté ϕ ou φ). Donc la description d'un problème est équivalente à la description d'une fonction.

Exemples de problèmes

— Additionner deux entiers

— Étant donné une liste d'étudiants et de leurs points d'examen, trouver les étudiants dont la moyenne des examens est supérieure ou égale à $12/20$.

— Déterminer si un polynôme à coefficients entiers a des racines entières.

1.3 Notion de programme

Un programme (ou algorithme) est une « procédure effective », c'est-à-dire exécutable par une machine. Par exemple un programme JAVA.

La compréhension du concept de programme est basée sur la compréhension du concept de « programme écrit dans un langage de programmation ».

Il existe plein de formalismes permettant la description d'une « procédure effective ».

1.4 Résultats principaux

1.4.1 Équivalence des langages de programmation

Existe-t-il des langages de programmation plus puissants que d'autres ? Tous les langages sont-ils équivalents ?

Il y a une équivalence entre langages en termes de calculabilité. Lorsqu'un problème est résolvable par un de ces langages, alors il l'est également par les autres. S'il existe un programme JAVAqui résout un problème, il existera aussi un programme C/C++, PHP,... qui peut résoudre le même problème. Il s'agit d'une équivalence théorique.

D'un point de vue théorique ces langages s'appellent des *langages complets*. Ils permettent tous de résoudre les mêmes problèmes. Un problème est donc calculable indépendamment du langage utilisé.

D'un point de vue pratique on peut voir des différences (le programme est plus court, ou s'écrit plus rapidement, ou s'exécute plus rapidement, ou est plus propre, ou plus fiable, etc). Certains langages sont donc mieux adaptés que d'autres pour certaines classes de problèmes.

1.4.2 Existence de problèmes non calculables

Il existe des problèmes qui ne peuvent être résolus par un programme et qui sont alors dit *non calculables*. Quelques exemples :

— détection de virus ;
— équivalence de programme ;
— déterminer si un polynôme à coefficients entiers a des racines entières, ...

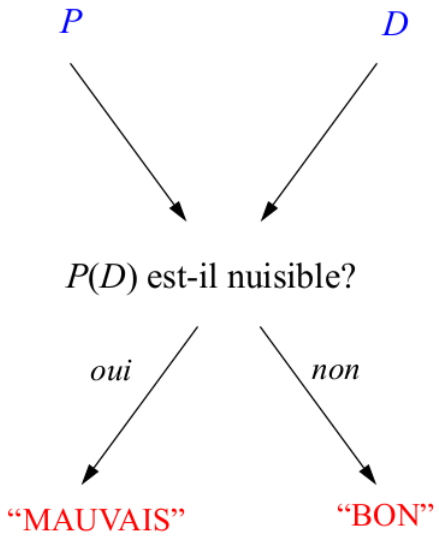
Exemple : Détection de virus informatique

On veut déterminer si un programme `P` avec une entrée `D` est nuisible.

Définition 1.1. *Un programme est dit* nuisible *si son exécution a pour effet de contaminer d'autres programmes (par exemple en se recopiant autre part). Dans certains cas, il peut également être considéré comme dangereux lorsqu'il rend la machine partiellement ou totalement inutilisable (par exemple un cheval de Troie contenant une bombe logique ou une fork bomb).*

Un bon antivirus serait alors un programme qui testerait des programmes en regardant s'ils se multiplient, tout en les empêchant de modifier (voir d'endommager) d'autres fichiers. Un tel programme, s'il existe, serait du même type que `detecteur(P, D)` avec `P` le programme à détecter et `D` un état du système de données dans lequel le programme s'exécuterait.

Spécification du programme `detecteur(P, D)` :
Préconditions : un programme `P` et une donnée `D`
Postconditions : `"Mauvais"` si `P(D)` est nuisible, `"Bon"` sinon.



Cependant il faut aussi que le détecteur ne soit pas nuisible pour que cette spécification soit valide. Or comme nous allons le démontrer, le détecteur ne peut être non nuisible.

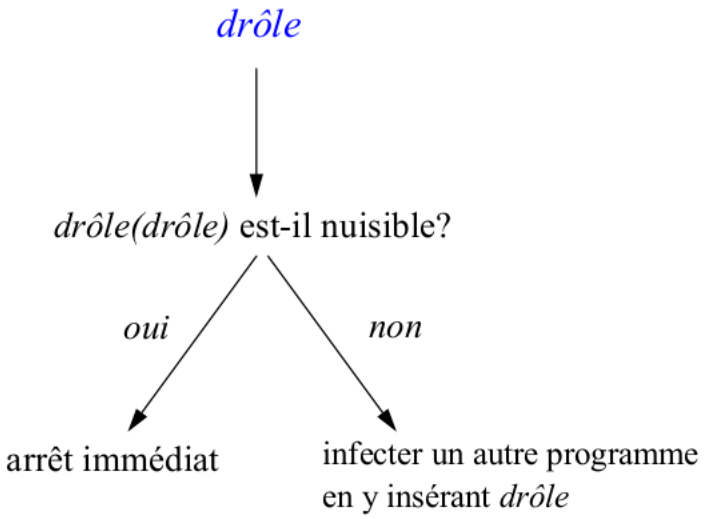
Théorème 1.2. *Il n'est pas possible d'écrire un programme antivirus non nuisible qui puisse déterminer avec sûreté si un programme* `P` *est nuisible avec une entrée* `D`*.*

Démonstration. Pour le prouver nous allons émettre l'hypothèse qu'un programme `detecteur(P, D)` existe et qu'il n'est pas nuisible. Dans ce cas, nous pouvons construire le programme `drole(P)` suivant :

```
drole(P)
    if detecteur(P,P) = "Mauvais"
        then stop
    else infecter un autre programme en y inserant P
```

Regardons maintenant si l'exécution de `drole(drole)`, c'est-à-dire si l'on donne le code source de `drole` à lui-même, est nuisible ou non.

```
drole(drole)
    if detecteur(drole, drole) = "Mauvais"
            then stop
    else infecter un autre programme en y inserant drole
```



— Supposons que `drole(drole)` soit nuisible. Lorsqu'on exécute, `drole(drole)`, `detecteur (drole,drole)` n'infecte rien car `detecteur` n'est pas nuisible.
Comme `detecteur` retourne `"Mauvais"`, le programme s'arrête. Rien n'a donc été infecté, ce qui est contradictoire avec le fait que `drole(drole)` soit nuisible.

— Si par contre il n'est pas nuisible, alors `detecteur(drole,drole)` ne va pas retourner `"Mauvais"` et le code de la clause **`else`** s'exécutera. Un autre programme est alors infecté ce qui contredit le fait que `drole(drole)` ne soit pas nuisible.

On a donc une contradiction dans tous les cas, ce qui implique que le programme `drole` ne peut pas exister. Vu que la seule hypothèse que nous avons formulée pour écrire le programme `drole` soit que le programme `detecteur` (non nuisible) existe, le programme `detecteur` (non nuisible) ne peut pas exister non plus. □

1.4.3 Existence de problèmes intrinsèquement complexes

Définition 1.3. *Un problème* intrinsèquement complexe *est un problème dont le meilleur algorithme n'a pas une complexité polynomiale, mais une complexité exponentielle ou pire. Peu importe les évolutions technologiques, un problème exponentiel ne peut et ne pourra être résolu que pour des données de petites tailles. Un exemple est le problème du voyageur de commerce qui doit rechercher le trajet le plus court pour relier n villes. Pour un problème intrinsèquement complexe, lorsqu'un ordinateur est n fois plus rapide, la taille des problèmes pouvant être résolus en temps raisonnable est incrémentée d'une petite valeur (voir le tableau ci-dessous).*

Supposons un ordinateur actuel sachant faire 100 millions d'instructions par secondes (100 Mips) et que le traitement d'un élément nécessite 100 instructions machines. Dénotons N_i la taille du « plus grand » exemple dont la solution peut-être calculée en 1 heure de temps calcul.

Complexité	Ordinateur actuel	100x plus rapide	1000x plus rapide
n	$N_1 = 3.6 \times 10^9$	$100N_1$	$1000N_1$
n^2	$N_2 = 6 \times 10^5$	$10N_2$	$31.6N_2$
n^3	$N_3 = 1530$	$4.64N_3$	$10N_3$
n^5	$N_4 = 81$	$2.5N_4$	$3.98N_4$
2^n	$N_5 = 31$	$N_5 + 6$	$N_5 + 10$
3^n	$N_6 = 20$	$N_6 + 4$	$N_6 + 6$

Ce tableau montre dès lors que, peu importe les évolutions technologiques, un problème intrinsèquement complexe **ne peut** et **ne pourra** être résolu que pour de petits exemples.

Cependant, dans la pratique, certains algorithmes résolvent efficacement les problèmes pratiques, et d'autres, également efficaces, permettent d'obtenir des **approximations** des solutions exactes.

1.5 Objectifs calculabilité et complexité

La Figure 1.1 illustre la frontière entre les problèmes non calculables (il n'existe pas un programme) et calculables (il existe un programme) dans l'univers des problèmes. Parmi les problèmes calculables il a ceux qui le sont en pratique (pas exponentiels) et ceux qui ne sont pas calculables en pratique.

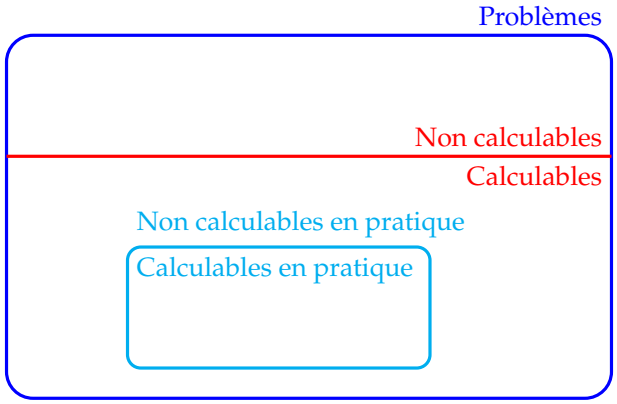


FIGURE 1.1 – Problèmes calculables / non calculables

L'intérêt de la calculabilité est de savoir quels problèmes sont calculables ou lesquels ne le sont pas. Si un problème est non calculable, il est alors inutile d'essayer de résoudre ce problème. Si le problème est calculable mais intrinsèquement complexe, il est inutile d'envisager un algorithme permettant de résoudre des problème de grande taille. Face à un problème non calculable ou intrinsèquement complexe, on peut essayer de relâcher le problème pour le rendre calculable ou de complexité polynomiale. Une solution à ce problème relâché peut par exemple être une approximation du problème initial.

Chapitre 2

Concepts

2.1 Ensembles, langages, relations et fonctions

2.1.1 Ensembles

Un *ensemble* est une collection d'objets, sans répétition, appelés les éléments de l'ensemble.

Notations

Ensemble vide : $\emptyset$

Ensemble fini : $\{0, 1, 2\}$, $\{00, 01, 10, 11\}$

Ensemble infini : $\{0, 1, 2, 3, 4, \dots\}$, $\{0, 1, -1, 2, -2, 3, -3, \dots\}$,
$\{a, aa, aaa, aaaa, \dots\}$, $\{a, b, aa, ab, ba, bb, aaa, aab, \dots\}$

Produit cartésien : $A \times B$

Le nombre d'éléments : $|A|$

Ensemble des sous-ensembles : 2^A ou $\mathcal{P}(A)$, e.g. $2^{\{2,4\}} = \{\{\}, \{2\}, \{4\}, \{2, 4\}\}$.
On peut remarquer que $|2^A| = 2^{|A|}$

Complément : $\overline{\mathrm{A}}$

2.1.2 Langages

Notations :

— Une *chaîne de caractères* ou un *mot* est une séquence **finie** de *symboles* (ou *caractères*).
Par exemple :

  — $abceced$,

  — 010101101

  — ♣♦♥♠✈❍❒

Les symboles peuvent être représentés par n'importe quel glyphe, cela n'a pas d'importance.

— Chaîne de caractères vide : ϵ

— Un *alphabet* Σ est un ensemble fini de symboles.
$\Sigma = \{1, 2\}$ $\Sigma = \{a, b, c\}$ $\Sigma = \{\cdot, \underline{\mathrm{A}}, a, 9\}$

— Un mot défini sur un alphabet est une séquence finie d'éléments de cet alphabet.

— Un *langage* est un ensemble de mots constitués de symboles d'un alphabet donné.

Ex : l'ensemble des palindromes définis sur $\{a, b\}$: $\epsilon, a, b, aa, aaa, aba, babaabab, aababbbabaa$

— L'ensemble de tous les mots possibles sur l'alphabet Σ : Σ^*

— $\Sigma = \phi, \quad \Sigma^* = \{\epsilon\}$

— $\Sigma = \{a\}, \quad \Sigma^* = \{\epsilon, a, aa, aaa, aaaa, \ldots\}$

— $\Sigma = \{0, 1\}, \quad \Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, \ldots\}$

2.1.3 Relations

Soient A, B des ensembles.

— Une *relation* R sur A, B est un sous-ensemble de $A \times B$. C'est-à-dire un ensemble de paires $\langle a, b\rangle$ avec $a \in A$, $b \in B$.

— Une relation est définie par sa table

— Elle peut s'écrire $\langle a, b\rangle \in R$ ou aRb ou $R(a, b)$.

2.1.4 Fonctions

Soient A, B des ensembles.

— Une *fonction* $f : A \to B$ est une relation telle que pour tout $a \in A$, il existe au plus un $b \in B$ tel que $\langle a, b\rangle \in f$.

— Écrire $f(a) = b$ est équivalent à $\langle a, b\rangle \in f$.

— S'il n'existe pas de $b \in B$ tel que $f(a) = b$ alors $f(a)$ est indéfini, $f(a) = \perp$. Le symbole $\perp$ est appelé *bottom*.

Propriétés des fonctions

Soit $f : A \to B$, on définit le *domaine* et l'*image* respectivement comme suit

$$\mathrm{dom}(f) = \{\, a \in A \mid f(a) \neq \perp \,\},$$
$$\mathrm{image}(f) = \{\, b \in B \mid \exists a \in A : b = f(a) \,\}.$$

Si $\mathrm{dom}(f) \subseteq A$, alors f est appelée *partielle* et si $\mathrm{dom}(f) = A$, alors f est appelée *totale*. Notez qu'avec cette définition, une fonction totale est partielle. Pour dire que $\mathrm{dom}(f) \subset A$, il faut dire que f n'est pas totale.

Une fonction est *surjective* si $\mathrm{image}(f) = B$ et *injective* si $\forall a, a' \in A : a \neq a' \implies f(a) \neq f(a')$.

Une fonction est *bijective* si elle est totale, injective et surjective.

On utilise aussi le concept d'*extension* : $f : A \to B$ est une extension de $g : A \to B$ si $\forall x \in A : g(x) \neq \perp \implies f(x) = g(x)$. Autrement dit, f a la même valeur que g partout où g est définie.

La fonction $\mathrm{sqrt}(x) = \sqrt{x}$ en mathématique n'est valable que pour $x \geq 0$. En revanche, un programme implémentant cette fonction, par exemple $\mathrm{mysqrt}(x) = \sqrt{x}$, retourne une valeur si $x \geq 0$ ou *Erreur* sinon. $\mathrm{mysqrt}(x)$ est une extension de la fonction $\mathrm{sqrt}(x)$ de base.

Définition d'une fonction

On définit une fonction par sa table qui peut être infinie.
On peut définir (la table d') une fonction de plusieurs façons :

— Par *un texte fini* déterminant sans contradiction ni ambiguïté le contenu de la table.

— Par un algorithme. Dans ce cas elle détermine la fonction *ainsi* qu'un moyen de la calculer.
Ex : $f(x) = 2x^3 + 5$

— En écrivant toutes les paires de la relation.

Il n'est toutefois pas nécessaire de décrire ou de connaître un moyen de la calculer pour pouvoir la définir sans ambiguïté ni contradiction.

Ex : $f(x) = \begin{cases} 1 & \text{s'il y a de la vie autre part que sur terre,} \\ 0 & \text{sinon.} \end{cases}$
(en supposant qu'il n'y ait pas d'ambiguïté sur le terme « vie »).

2.2 Ensemble énumérable

Quelle est la taille d'un ensemble ? Comment comparer la taille d'ensembles infinis ?

Avant de dire ce qu'est un ensemble énumérable, on doit savoir que deux ensembles ont le même cardinal s'il existe une bijection entre eux.

Définition 2.1 (Ensemble énumérable)**.** *Un ensemble est* énumérable *ou* dénombrable *s'il est fini ou s'il a le même cardinal que* $\mathbb{N}$*.*

Une autre manière plus intuitive de voir si un ensemble infini est énumérable est de voir s'il existe une énumération de ses éléments.

Propriété 2.2. *Un ensemble infini E est énumérable s'il existe une liste de ses éléments*

$$e_0, e_1, e_2, \ldots, e_n, \ldots$$

Cette liste doit contenir tous les éléments de cet ensemble, sans répétition. La bijection est alors la fonction $f(i) = e_i$.

2.2.1 Exemples

Exemple 2.3. *L'ensemble des entiers $\mathbb{Z} = \{0, -1, 1, 2, -2, \ldots\}$.*

Démonstration. On peut avoir une bijection entre $\mathbb{Z}$ et $\mathbb{N}$, ils peuvent être énumérés et on peut donner un numéro pour chaque élément, il existe donc une énumération.

$\mathbb{N}$	0	1	2	3	4	5	6	…
$\mathbb{Z}$	0	-1	1	-2	2	-3	3	…

□

Exemple 2.4. *L'ensemble des nombres pairs.*

Démonstration. Le principe reste le même : on peut établir une bijection entre l'ensemble des nombres pairs et $\mathbb{N}$.

$\mathbb{N}$	0	1	2	3	4	5	6	…
Nombres pairs	0	2	4	6	8	10	12	…

□

Exemple 2.5. *L'ensemble des paires d'entiers, des triplets, ...*

Démonstration. Il existe une énumération, il existe une bijection comme celle-ci :

$\mathbb{N}$	0	1	2	3	4	5	...
Ensemble des paires d'entiers	0,0	1,0	0,1	0,2	2,0	1,2	...
Ensemble des triplets d'entiers	0,0,0	0,0,1	0,1,0	1,0,0	0,1,1	1,0,1	...

Pour énumérer les paires d'entiers, on énumère les paires de somme 0, les paires de sommes $1, \ldots$ De même avec les triplets, quadruplets, ...
Dans le cas des paires d'entiers, cela revient à construire un tableau à deux dimensions de toutes les paires, et à parcourir ce tableau en suivant les différentes diagonales montantes. □

Exemple 2.6. *Les rationnels, même s'ils ont une représentation décimale infinie, peuvent être représentés de manière finie en fraction d'entiers.*

Démonstration. Comme tout rationnel peut s'écrire sous la forme d'un numérateur et d'un dénominateur, un rationnel est équivalent à une paire d'entiers (cette équivalence est univoque en prenant un numérateur et un dénominateur premiers entre eux, un dénominateur toujours positif et la paire $(0, 1)$ pour le nombre 0). On peut donc faire une démonstration similaire à celle des paires d'entiers pour prouver que les rationnels sont en bijection avec l'ensemble $\mathbb{N}$ (quitte à ne pas prendre certaines paires qui ne correspondent pas à un rationnel). □

Exemple 2.7. *L'ensemble des sous-ensembles finis d'entiers.*

Démonstration. Par l'exemple 2.5, les ensembles des sous-ensembles de tailles 2, 3, 4, 5,... sont tous énumérables. On peut alors créer un tableau reprenant tous les sous-ensembles finis possibles et dans lequel la ligne n comprend tous les sous-ensembles de taille n. Ces ensembles sont en bijection avec les naturels ($\mathbb{N}$).

$\mathbb{N}$	0	1	2	3	4	5	6
sous-ensembles taille 0							...
sous-ensembles taille 1	0	1	2	3	4	5	...
sous-ensembles taille 2	0,0	1,0	0,1	0,2	2,0	1,2	...
sous-ensembles taille 3	0,0,0	0,0,1	0,1,0	1,0,0	0,1,1	1,0,1	...
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

□

Exemple 2.8. *L'ensemble des chaînes finies de caractères sur un alphabet fini.*

Démonstration. Variante de la démonstration précédente, la technique reste la même (l'alphabet admet k éléments au lieu de 10). □

Exemple 2.9. *L'ensemble des fonctions de $\{0, 1\}$ vers $\mathbb{N}$ est énumérable.*

Démonstration. Une fonction de $\{0, 1\}$ vers $\mathbb{N}$ peut être représentée par deux couples de deux entiers $(0, f(0))$ et $(1, f(1))$, donc est un quadruple. L'ensemble des fonctions de $\{0, 1\}$ vers $\mathbb{N}$ est équivalent à l'ensemble des quadruples et est donc énumérable. □

Exemple 2.10. *L'ensemble des programmes Java.*

Démonstration. Un programme Java est une séquence finie d'un alphabet fini (ASCII étendu). Donc il y a potentiellement autant de programmes qu'il y a de naturels, ni plus ni moins, ce qui nous donne une bijection entre les programmes et $\mathbb{N}$. □

Exercice 2.11. *Un nombre réel x est appelé algébrique si x est la racine d'un polynôme d'équation $a_0 + a_1x + a_2x^2 + ... + a_nx^n$. avec a_i entier*
Démontrer que x est énumérable.

2.2.2 Propriétés

Propriété 2.12. *Tout sous-ensemble d'un ensemble énumérable est énumérable.*

Démonstration. Soit A un sous-ensemble (infini) d'un ensemble E énumérable. Comme E est énumérable, il existe une énumération de ses éléments

$$e_0, e_1, \ldots, e_n, \ldots$$

Enlevons de cette liste les éléments qui ne sont pas dans A. On obtient une nouvelle liste qui énumère les éléments de A. □

Propriété 2.13. *L'union et l'intersection de deux ensembles énumérables sont énumérables.*

Démonstration. Soient A et B deux ensembles énumérables infinis (le cas fini étant trivial). Leur intersection est énumérable car elle est un sous-ensemble de A, par la propriété précédente. Les éléments des ensembles A et B peuvent être listés

$$A = \{a_0, a_1, \ldots, a_n, \ldots\} \quad B = \{b_0, b_1, \ldots, b_n, \ldots\}$$

Les éléments de $A \cup B$ peuvent être également listés

$$a_0, b_0, a_1, b_1, \ldots, a_n, b_n, \ldots$$

en y retirant ensuite les doublons. Donc, $A \cup B$ est énumérable. □

Propriété 2.14. *L'union d'une infinité énumérable d'ensembles énumérables est énumérable.*

Démonstration. La démonstration est similaire à la démonstration de l'énumérabilité de $\mathbb{Q}$. On met chaque ensemble en ligne. Il y a un nombre énumérable de lignes qu'on peut numéroter par des naturels. On énumère l'union des ensembles en parcourant le tableau en diagonale montante en partant de $(0, 0)$. □

Propriété 2.15. *Tout ensemble de chaînes finies de caractères sur un alphabet fini est énumérable.*

Démonstration. C'est un sous-ensemble de l'ensemble de toutes les chaînes finies de caractères, qui est énumérable par l'exemple 2.8. □

Propriété 2.16. *L'ensemble des programmes Java est énumérable.*

Démonstration. Voir exemple 2.10. □

Propriété 2.17. *Tout langage (avec un alphabet fini) est énumérable.*

Démonstration. Un langage est un ensemble de chaînes finies de caractères d'un alphabet fini. Il est donc énumérable. □

Propriété 2.18. *L'union d'une infinité non énumérable d'ensembles énumérables peut ne pas être énumérable.*

Démonstration. Par exemple, l'union des singletons $\{x\}$ pour tout réel x forme l'ensemble des réels $\mathbb{R}$ qui n'est pas énumérable :

$$\bigcup_{x \in \mathbb{R}} \{x\} = \mathbb{R}.$$

C'est un contre-exemple. □

Remarque 2.19. *Une bonne intuition à avoir : tout ensemble dont les éléments peuvent être représentés de manière finie est énumérable.*

Dans le cours, lorsqu'on devra montrer qu'un ensemble est énumérable, les techniques suivantes pourront être utilisées :

— montrer qu'il y a une bijection avec $\mathbb{N}$;
— montrer que l'ensemble est fini ;
— écrire un programme qui énumère l'ensemble ;
— utiliser une des propriétés ci-dessus.

2.3 Cantor

On va montrer qu'il existe des ensembles non énumérables par diagonalisation. Par exemple, l'ensemble des réels, $\mathbb{R}$.

Exemple 2.20. *Exemple de démonstration par diagonalisation :*

1. *On construit une table, dans laquelle on fait l'hypothèse qu'on a réussi à lister* TOUS *les grands mathématiciens.*

D	*E*	*M*	*O*	*R*	*G*	*A*	*N*		
A	**B**	*E*	*L*						
B	*O*	**O**	*L*	*E*					
B	*R*	*O*	**U**	*W*	*E*	*R*			
S	*I*	*E*	*R*	**P**	*I*	*N*	*S*	*K*	*I*
W	*E*	*I*	*E*	*R*	**S**	*T*	*R*	*A*	*S*	*S*

2. *Sélectionner la diagonale :* $\mathrm{diag} = DBOUPS$
3. *Modifier l'élément égal à la diagonale :* $\mathrm{diag}' = CANTOR$ *(prendre à chaque fois la lettre précédente dans l'alphabet)*
4. *Montrer que l'élément n'est pas dans la liste* $\implies$ *Contradiction*
5. *Conclusion :*
   — *Soit on sait que la liste est complète*
   $\implies$ *CANTOR n'est pas un grand mathématicien (cas utilisé pour démontrer halt).*
   — *Soit on sait que CANTOR est un grand mathématicien*
   $\implies$ *la liste est incomplète (cas utilisé pour la diagonalisation de CANTOR)*

Théorème 2.21 (Diagonalisation de Cantor)**.** *Soit* $E = \{x \text{ réel } | 0 < x \leq 1\}$*, E est non énumérable.*

Démonstration. Démonstration par l'absurde. En supposant que E soit énumérable, on va montrer qu'un nombre d' n'est pas dans l'énumération alors qu'on sait que d' est un nombre réel compris entre 0 et 1.

On suppose E énumérable. Donc il existe une énumération des éléments de E. Soit $x_0, x_1, \ldots, x_k, \ldots$ cette énumération. On peut représenter un nombre x_k comme étant une suite de chiffres x_{ki} : $x_k = 0.x_{k0}x_{k1}\ldots x_{kk}\ldots$ [1]

1. On peut donc construire une table infinie :

	1 digit	2 digit	3 digit	…	k+1 digit	…
x_0	x_{00}	x_{01}	x_{02}	…	x_{0k}	…
x_1	x_{10}	x_{11}	x_{12}	…	x_{1k}	…
x_2	x_{20}	x_{21}	x_{22}	…	x_{2k}	…
⋮	⋮	⋮	⋮	⋱	⋮	⋮
x_k	x_{k0}	x_{k1}	x_{k2}	…	x_{kk}	…
⋮	⋮	⋮	⋮	⋮	⋮	⋱

1. Certains nombres ont deux écritures décimales : par exemple, $0.999\ldots = 1.000\ldots$. Il suffit d'en choisir une des deux pour que l'écriture décimale d'un nombre soit définie de façon univoque.

2. Sélection de la diagonale (celle-ci est un nombre réel compris entre 0 et 1)

$$d = 0.x_{00}x_{11}\dots x_{kk}\dots$$

3. Modification de cet élément d pour obtenir

$$d' = 0.x'_{00}x'_{11}\dots x'_{kk}\dots$$

Où

$$x'_{ii} = 5 \text{ si } x_{ii} \neq 5 \quad \text{et} \quad x'_{ii} = 6 \text{ si } x_{ii} = 5$$

On a toujours que cet élément d' est un réel compris entre 0 et 1 [2].

4. Le nombre d' est dans l'énumération, car E est énumérable (par hypothèse). Il existe donc p tel que $x_p = d'$,

$$x_p = 0.x_{p0}x_{p1}\dots x_{pp}\dots$$
$$=$$
$$d' = 0.x'_{00}x'_{11}\dots x'_{pp}\dots$$

Il y a une contradiction car $x'_{pp} \neq x_{pp}$ par la définition de x'_{pp}. Donc $d' \neq x_p$ ce qui implique que d' n'est pas dans l'énumération.

5. Conclusion : E n'est pas énumérable.

□

2.3.1 Exemples

Quelques ensembles non énumérables :

Exemple 2.22. *L'ensemble* $\mathbb{R}$

Démonstration. Voir ci-dessus. □

Exemple 2.23. *L'ensemble des sous-ensembles de* $\mathbb{N}$, $\mathcal{P}(\mathbb{N})$

Démonstration. Rappelons-nous tout d'abord que comme $\mathbb{N}$ est infini, ses sous-ensembles peuvent l'être aussi. Il est adéquat de visualiser un ensemble à l'aide d'un mot binaire où le bit i vaut 1 si le $i^{\text{ième}}$ élément est pris dans le sous-ensemble. Par exemple, le sous-ensemble $\{1, 4\}$ de $\{1, 3, 4\}$ peut être représenté par 101. Seulement, comme il y a un nombre infini de nombre naturels, on a mot binaire infini. Par exemple, les nombres pairs, c'est le mot $10101010101\dots$:

```
0123456789...
1010101010... nombres pairs

0123456789...
0011010100... nombres premiers

0123456789...
0001001001... multiples de 3 non-nuls
```

On voit maintenant la bijection entre les sous-ensembles de $\mathbb{N}$ et $[0, 1]$. En effet, chaque suite m de 0 et de 1 peut être associée à l'écriture binaire $0.m$, qui correspond à un réel dans $[0, 1]$ [3]. Par exemple, on associe l'ensemble des nombres au réel $0.101010101\dots = 2^{-1} + 2^{-3} + 2^{-5} + \dots = 2/3$. □

2. De plus, puisqu'il ne contient que des 5 et des 6 il n'a qu'une écriture décimale, donc notre choix quant à l'écriture décimale n'a pas d'influence et il appartient forcément au tableau.

3. Comme dans la diagonalisation de Cantor, il faut aussi s'occuper des cas où un réel a plusieurs écritures binaires pour avoir une bijection parfaite. Ces détails techniques ne sont pas importants dans ce cours.

Exemple 2.24. *L'ensemble des chaînes infinies de caractères sur un alphabet fini.*

Démonstration. On utilise le même raisonnement que pour les sous-ensembles de $\mathbb{N}$ sauf que pour un alphabet de k symboles, on utilise la représentation en base k des réels. □

Exemple 2.25. *L'ensemble des fonctions de $\mathbb{N}$ dans $\mathbb{N}$ (Cas important) et de $\mathbb{N}$ dans $\{0,1\}$.*

Démonstration. Si c'était énumérable, soit $f_0, f_1, f_2, \ldots$ leur dénombrement. On construit le tableau

$$\begin{array}{ccccc} f_0(0) & f_0(1) & f_0(2) & f_0(3) & \cdots \\ f_1(0) & f_1(1) & f_1(2) & f_1(3) & \cdots \\ f_2(0) & f_2(1) & f_2(2) & f_2(3) & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{array}$$

et on conclut par diagonalisation de façon semblable à $\mathbb{R}$ en construisant $d\colon \mathbb{N} \to \mathbb{N}$ tel que $d(k) = f_k(k) + 1$. On a ainsi une fonction $d(x)$ qui ne se trouve pas dans le tableau. □

Dans le cours, lorsqu'on devra montrer qu'un ensemble est non énumérable, les techniques suivantes pourront être utilisées

— montrer qu'il y a une bijection avec $\mathbb{R}$
— utiliser la diagonalisation (cf. Cantor)

Exercice 2.26. *L'ensemble des fonctions de $\mathbb{N}$ vers $\{0,1\}$ est-il énumérable ?*

Exercice 2.27. *Un nombre réel x est appelé transcendantaux si x n'est pas algébrique. (Un nombre réel x est appelé algébriques si x est la racine d'un polynôme d'équation $a_0 + a_1x + a_2x^2 + ... + a_nx^n$. avec a_i entier). Démontrer que x est non-énumérable s'il est transcendantaux.*

2.3.2 Au delà de l'énumérable

On a d'abord parlé d'ensembles finis, ensuite nous avons introduit le concept d'infini énumérable qui est plus grand que tous les ensembles finis. Mais il existe plus grand que l'infini énumérable. L'ensemble des réels $\mathbb{R}$, l'ensemble des fonctions de $\mathbb{N}$ dans $\mathbb{N}$, ainsi que l'ensemble des sous-ensemble de $\mathbb{N}$ ($2^{\mathbb{N}}$) sont non énumérables et ont la même taille. Ils ont la puissance du continu.

Peut-on encore aller plus loin ? Oui, il suffit de considérer l'ensemble des fonctions de $\mathbb{R}$ dans $\mathbb{R}$, ainsi que l'ensemble des sous-ensembles de $\mathbb{R}$ ($2^{\mathbb{R}}$). Ces ensembles ne peuvent pas être mis en bijection avec $\mathbb{R}$.

Encore plus loin ? Oui, en prenant l'ensemble des sous-ensembles de l'ensemble défini juste avant (on peut montrer que pour tout ensemble E, $|E| < |\mathcal{P}(E)| = 2^{|E|}$). On obtient alors $2^{2^{\mathbb{R}}}$, et ainsi de suite…

On peut ainsi schématiser la taille des ensembles comme ceci :

$$|\emptyset| < |\{1,2,3\}| < |\mathbb{N}| < |2^{\mathbb{N}}| = |\mathbb{R}| < |2^{\mathbb{R}}| < |2^{2^{\mathbb{R}}}| < |2^{2^{2^{\mathbb{R}}}}| \ldots$$

Est-ce qu'il existe encore plus grand ? Oui, effectivement, l'union de tous ces ensembles est plus grand que chacun de ces ensembles [4].

4. Pour aller plus loin : en théorie des ensembles ZFC, cette construction s'écrit avec la lettre $\beth$ (prononcé **beth**), telle que $\beth_0 = \mathbb{N}, \beth_1 = 2^{\mathbb{N}} = \mathbb{R}, \ldots$. L'union de tous ces ensembles est plus grande que chacun d'eux, et se note $\beth_{\mathbb{N}}$. On peut continuer la construction à partir de là, et obtenir des ensembles encore plus grands.

Par ailleurs, l'ensemble de *tous* les cardinaux est noté avec la lettre $\aleph$ (prononcé **aleph**), avec $\aleph_0 = |\mathbb{N}|$. On pourrait être tenté de prendre l'ensemble de tous les cardinaux pour avoir un ensemble encore plus grand. Mais ce ne serait plus un ensemble (c'est une classe dans la théorie des ensembles NBG) !

Finalement, l'hypothèse du continu stipule que $\beth_1 = \aleph_1$, c'est-à-dire qu'il n'y a pas de cardinal entre $\mathbb{N}$ et $\mathbb{R}$. Ce n'est qu'en 1963 qu'on s'est rendu compte qu'elle était en fait indépendante de la théorie des ensembles ZFC : on peut supposer qu'elle soit vrai ou qu'elle soit fausse sans obtenir de contradiction.

2.4 Conclusion

Les ensembles énumérables sont importants pour la suite du cours. En informatique, on ne considère que les ensembles énumérables. En effet, on constate que l'univers des programmes (Java ou autres) est énumérable (voir propriété 2.16), alors que l'univers des problèmes (fonctions de $\mathbb{N}$ dans $\mathbb{N}$) est non énumérable (voir exemple 2.25). On ne peut donc forcément pas faire correspondre un programme à chaque problème et on en conclut donc qu'un grand nombre de problèmes ne sont pas calculables.

Chapitre 3

Résultats fondamentaux

3.1 Algorithmes et effectivité

Qu'est-ce qu'un algorithme ?

Définition 3.1 (Algorithme)**.** *C'est une procédure qui peut être appliquée à n'importe quelle donnée et qui a pour effet de produire un résultat. C'est un ensemble fini d'instructions qui peuvent être exécutées. Dans cette partie du cours, on ne prend pas en compte la taille des données, des instructions ni de la mémoire disponible, mais on les considère comme finies.*

Remarque 3.2. *Un algorithme n'est pas une fonction, mais un algorithme calcule une fonction ou une extension de fonction. De plus dans le cours on se limite aux fonctions de $\mathbb{N}^n$ dans $\mathbb{N}$. En effet cela revient au même de considérer les fonctions de $\mathbb{N}^n$ dans $\mathbb{N}^n$ ($\mathbb{N}^n$ est énumérable, il existe donc une bijection avec $\mathbb{N}$). On va aussi utiliser Java comme modèle étant donné que c'est plus facile et qu'on va montrer que les modèles complets sont équivalents.*

3.2 Fonctions calculables, ensembles récursifs et récursivement énumérables

3.2.1 Fonction calculable

Définition 3.3 (Fonction calculable)**.** *Une fonction f est calculable s'il existe un algorithme qui, recevant comme donnée n'importe quels nombres naturels $x_1, \ldots, x_n$ fournit **tôt ou tard** comme résultat $f(x)$ s'il existe.*

S'il ne se termine pas c'est que $f(x) = \perp$.

Remarque 3.4. *Il faut faire la distinction entre la non-existence d'un algorithme et ne pas être capable de le trouver. (Voir exemple TP et cours : y a-t-il une rose verte sur Mars ou encore x occurrences de 5 dans π).*

Remarque 3.5. *Une fonction peut-être totale calculable ou partielle calculable.*

3.2.2 Ensemble récursif et récursivement énumérable

Soit $A \subseteq \mathbb{N}$

Définition 3.6 (Ensemble récursif)**.** *Un ensemble A est récursif s'il existe un algorithme qui recevant un $x \in \mathbb{N}$, fournit **tôt ou tard** comme résultat* $\begin{matrix} 1 \text{ si } x \in A \\ 0 \text{ si } x \notin A \end{matrix}$ *. L'algorithme décide si x est dans A ou non.*

Définition 3.7 (Ensemble récursivement énumérable)**.** *Un ensemble A est récursivement énumérable ($\in RE$) s'il existe un algorithme qui recevant un $x \in \mathbb{N}$,*

— *fourni **tôt ou tard** comme résultat 1 si $x \in A$.*

— *ne se termine pas ou retourne un résultat $\neq 1$ si $x \notin A$.*

Définition 3.8 (Ensemble co-récursivement énumérable)**.** *Un ensemble A est* co-récursivement énumérable *si son complément $\bar{A} = \mathbb{N} \setminus A$ est récursivement énumérable.*

Remarque 3.9. *Le souci avec les ensembles récursivement énumérables, c'est que même si l'algorithme permet de dire si x est dans A, si x n'est pas dans A on ne sait rien dire. En effet, on ne peut pas dire à un moment qu'on arrête l'algorithme et que l'élément n'est pas dans A. Car il est possible que x soit dans A et que l'algorithme ne l'ait pas encore déterminé (l'algorithme retourne tôt ou tard un résultat si x est dans A !).*

Définition importante :

Définition 3.10 (Fonction caractéristique)**.** *Fonction caractéristique de A : $X_A \colon \mathbb{N} \to \{0, 1\}$, tel que*

$$X_A(x) = \begin{cases} 1 & \text{si } x \in A \\ 0 & \text{si } x \notin A \end{cases}$$

Propriété 3.11. *A est un ensemble récursif ssi X_A est une fonction totale calculable.*

On remarque en effet que X_A décide si x appartient à A ou non.

Propriété 3.12. *A est un ensemble récursivement énumérable ssi $A = \mathrm{dom}(f)$ où f est une fonction (totale ou partielle) calculable.*

Démonstration. Supposons que f est calculable avec $A = \mathrm{dom}(f)$. Alors il existe un programme P_f qui calcule f. On peut construire un programme Q qui avec input x appelle $P_f(x)$, puis renvoie 1. Si $x \in A$, $P_f(x)$ se termine, $Q(x)$ renverra 1. Sinon, $P_f(x)$ ne se terminera pas ($P_f(x) = \perp$), donc $Q_f(x)$ non plus. On en déduit que A est récursivement énumérable.

Réciproquement si A est récursivement énumérable, il existe un programme Q_A qui renvoie 1 ssi $x \in A$. La restriction à A de la fonction calculée par Q_A est calculable est telle que $\mathrm{dom}(f) = A$. □

Propriété 3.13. *A est un ensemble récursivement énumérable ssi*

— *A est vide ou*

— *$A = \mathrm{image}(f)$ et f est une fonction totale calculable.*

Démonstration. D'une part si A est vide, le programme renvoyant toujours $\perp$ détermine A, donc A est récursivement énumérable. D'autre part si $A = \mathrm{image}(f)$ avec f calculable, on montre que A est récursivement énumérable. À l'aide du programme P_f qui calcule f, on appelle $P_f(0)$, $P_f(1)$, $P_f(2)$, ... (en les lançant dans des threads, à l'aide d'une diagonale montante par exemple) jusqu'à ce que $P_f(k) = x$ où on retourne 1. Si $x \in A = \mathrm{image}(f)$, on retournera 1 tôt ou tard. Sinon, on ne terminera jamais, ce qui équivaut à retourner $\perp$.

Réciproquement, supposons que A soit récursivement énumérable. Si A est vide, l'implication est vérifiée. Si f_A est la fonction qui détermine A, soit P le programme qui appelle $f_A(0)$, $f_A(1)$, ... en même temps (en les lançant dans des threads). $P(n)$ renvoie le numéro du thread étant le n-ième à s'être arrêté et à avoir renvoyé 1. Ainsi, la fonction totale construit par P énumère A, c'est-à-dire $\mathrm{image}(f) = A$. □

Propriétés importantes

Propriété 3.14. *A récursif $\Rightarrow$ A récursivement énumérable. (Propriété plus faible)*

Propriété 3.15. *A récursif $\Rightarrow$ $\overline{A}$ récursif.*
On peut facilement créer un programme qui retourne 1 - le programme qui décide A.

Propriété 3.16. *Si A est récursivement énumérable et co-récursivement énumérable ($\overline{A}$ récursivement énumérable) alors A est récursif.*

Démonstration. Il suffit de créer un programme Q qui, pour décider récursivement si $x \in A$, appelle le programme qui décide A et le programme qui décide $\overline{A}$ dans deux threads et renvoie 1 si $x \in A$ et 0 si $x \in \overline{A}$. En effet, on a $x \in A$ ou $x \in \overline{A}$. Dès lors, si $x \in A$ alors le programme qui décide A se finira, et si $x \in \overline{A}$ alors c'est l'autre programme qui se terminera. A est donc bien un ensemble récursif. □

Propriété 3.17. *A fini $\Rightarrow$ A récursif.*

Démonstration. Si $A = a_0, \ldots, a_n$, alors le programme suivant énumère tous ces éléments

if $x = a_0$ **then**
  print 1
else if $x = a_1$ **then**
  print 1
  ...
else if $x = a_n$ **then**
  print 1
else
  print 0
end if

□

Propriété 3.18. *$\overline{A}$ fini $\Rightarrow$ A récursif. (Comme propriété précédente)*

Exemple 3.19. *La fonction $f(x)$ qui renvoie 1 s'il existe au moins x occurrences successives de 5 dans π est-elle calculable ?*

Oui. Si π est un nombre univers comme certains en font l'hypothèse. Les décimales de π contiendrait alors toutes les suites possibles et imaginables de chiffres, par exemple x 5 successifs. On écrit alors simplement l'algorthme suivant.

print 1

Si cependant π n'est pas un nombre univers, alors il y a maximum **a** *occurrences successives de 5 dans π et on écrit l'algorithme suivant.*

if $x \leq$ **a** ***then***
  print 1
else
  print 0
end if

Si on remplace "au moins" par "exactement", est-ce toujours calculable ? Oui, il suffit de remplacer $\leq$ par $=$ dans le second algorithme. Dans tous les cas, un algorithme fournissant $f(x)$ comme résultat existe, bien que nous ne savons exactement pas duquel il s'agit.

Propriété 3.20. *Tout programme qui a un nombre fini d'inputs différents est calculable : on fait comme dans la propriété 40, mais en remplaçant les 1 par ce que doit renvoyer la fonction pour tel input (ça ne marche pas s'il est infini car le taille du code du programme doit être fini donc on ne peut pas tout hardcoder).*

3.3 Thèse de Church-Turing

Grâce à sa définition, on sait montrer qu'une fonction est calculable. Mais comment montrer qu'une fonction n'est pas calculable ? Pour ça on a besoin d'une définition couvrant la totalité des fonctions calculables.

On peut également se demander si prendre un modèle particulier est restrictif ? Un modèle peut être la logique mathématique (Gödel), ou la machine de Turing, ou les langages de programmation, etc.

La thèse de Turing répond à ces questions.

1. Aucun modèle de la notion de fonction calculable n'est plus puissant que les Machines de Turing. Version moderne : Une fonction est calculable s'il existe un programme d'ordinateur qui calcule cette fonction.
2. Toute fonction calculable est calculable par une machine de Turing.
3. Toutes les définitions formelles de la calculabilité connues à ce jour sont équivalentes (théorème, ça a été démontré) (justifie l'utilisation de Java comme modèle).
4. Toutes les formalisations de la calculabilité établies par la suite seront équivalentes aux définitions connues.

1, 2 et 4 sont des thèses universellement reconnues comme vraies et la 3 est un théorème.

3.4 Programmes et fonctions

Dans le cours on utilise un langage de programmation, Java, comme modèle.

Définition 3.21 (P)**.** *Soit P l'ensemble des programmes Java qui reçoivent un ou plusieurs entiers comme donnée(s) et qui impriment/retournent un résultat.*

Propriété 3.22. *P est un ensemble infini dénombrable, car c'est la chaîne de caractères d'un alphabet fini. P est récursif, car il existe un programme, le compilateur, qui détermine si une chaîne de caractères est un programme Java ou non.*

Définition 3.23 (Énumération de P)**.** *$P = P_0, P_1,\ldots,P_k,\ldots$ est l'énumération des programmes Java sans répétition. Ce qui implique qu'on peut numéroter les programmes Java.*

Définition 3.24 (P_k)**.** *P_k est le programme numéro k dans P.*

Définition 3.25 ($\varphi_k^{(n)}$)**.** *$\varphi_k^{(n)} : \mathbb{N}^n \to \mathbb{N}$ est la fonction calculée par P_k.*

Remarque 3.26. *$P_{12} \neq P_{47}$ mais ça ne veut* pas *dire que $\varphi_{12} \neq \varphi_{47}$. En effet, il existe une infinité de manière de coder une fonction.*

Propriété 3.27. *Il existe donc une fonction $f : \mathbb{N} \to P$ telle que :*

— *$f(k) = P_k$;*
— *f calculable ;*
— *k (numéro d'un programme) et P_k sont deux représentations distinctes d'un même objet.*

3.5 Existence de fonctions non calculables

Il existe beaucoup de fonctions non calculables, car le nombre de fonctions de $\mathbb{N}$ dans $\mathbb{N}$ est non dénombrable (Exemple 2.25). Or le nombre de programmes Java est dénombrable. Donc il y a beaucoup de fonctions qui ne sont pas calculables.

On ne va s'intéresser qu'aux fonctions qui sont définies par une table finie ou une table infinie que l'on peut décrire de façon finie (on n'est pas capable d'écrire une définition infinie). Il en existe une infinité dénombrable.

On va maintenant montrer que ce n'est pas parce que la table d'une fonction est définie de manière finie que la fonction est nécessairement calculable (exemple la fonction halt qui détermine si un programme se termine ou non).

3.6 Problème de l'arrêt

Définition 3.28 (halt). *halt est la fonction : $P \times \mathbb{N} \to \mathbb{N}$ telle que*
halt(n, x) = 1 si $P_n(x)$ se termine
halt(n, x) = 0 sinon
ce qui équivaut à
halt(n, x) = 1 si $\varphi_n(x) \neq \perp$
halt(n, x) = 0 sinon

Propriété 3.29. *halt est une fonction bien définie, totale et sa table est infinie, mais décrite de manière finie. Or on va montrer que halt n'est pas calculable par diagonalisation (comme pour la démonstration de Cantor).*

Remarque 3.30. *Attention, juste dire qu'on ne sait pas écrire un programme qui calcule halt ne prouve pas qu'on ne sait pas calculer halt. En effet, comme vu dans le chapitre 1, prouver que le programme existe est différent de savoir écrire le programme.*

Théorème 3.31 (halt). *halt n'est pas calculable.*

Démonstration. On suppose halt calculable.

1. On peut donc construire une table infinie définissant la fonction halt :

	0	1	2	$\cdots$	k	$\cdots$
p_0	$halt(0,0)$	$halt(0,1)$	$halt(0,2)$	$\cdots$	$halt(0,k)$	$\cdots$
p_1	$halt(1,0)$	$halt(1,1)$	$halt(1,2)$	$\cdots$	$halt(1,k)$	$\cdots$
p_2	$halt(2,0)$	$halt(2,1)$	$halt(2,2)$	$\cdots$	$halt(2,k)$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
p_k	$halt(k,0)$	$halt(k,1)$	$halt(k,2)$	$\cdots$	$halt(k,k)$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

2. Sélection de la diagonale

$$diag : halt(0,0), halt(1,1), \cdots, halt(k,k), \cdots$$

$$diag(n) = halt(n,n)$$

3. Modification de cet élément diag pour obtenir $diag'(n) = \begin{matrix} 1 \text{ si halt(n,n) = 0} \\ \perp \text{ si halt(n,n) = 1} \end{matrix}$
$diag'$ est calculable, car halt est calculable (il y a moyen d'écrire un programme qui calcule $diag'$ en utilisant $diag$).
4. Contradiction :
Donc il existe un programme P_d qui calcule $diag'$: $diag'(d) = \begin{matrix} 1 \text{ si halt(d,d) = 0} \\ \perp \text{ si halt(d,d) = 1} \end{matrix}$ Mais,

— Si $diag'(d) = 1$
$\Rightarrow$ $halt(d,d) = 0$
$\Rightarrow$ $P_d(d)$ ne se termine donc pas
$\Rightarrow$ $diag'(d)$ ne se termine pas
$\Rightarrow$ $diag'(d) = \perp$ or on a supposé que $diag'(d) = 1$
$\Rightarrow$ Contradiction.

— Si $diag'(d) = \perp$
$\Rightarrow$ $halt(d, d) = 1$
$\Rightarrow$ $P_d(d)$ se termine
$\Rightarrow diag'(d)$ termine
$\Rightarrow diag'(d) = 1$ or on a supposé que $diag'(d) = \perp$
$\Rightarrow$ Contradiction.

5. Conclusion : $diag'$ n'est pas calculable $\Rightarrow$ $diag$ n'est pas calculable $\Rightarrow$ halt n'est pas calculable.

□

Conclusion Il n'existe pas d'algorithme qui détermine si n'importe quel programme P_n se termine ou non. Mais dans certains formalismes qui ne sont pas des modèles complets la fonction halt de ce formalisme est calculable. Par exemple un langage qui ne permet de calculer que des fonctions totales (exemple java sans aucune boucle), halt est calculable (halt retourne toujours 1).

De plus, il faut faire attention, car ce n'est pas parce que halt n'est pas calculable que pour un programme donné k, $halt(k, x)$ est non calculable. Par exemple, si P_{32} se termine pour tout input, alors $halt(32, x)$ est calculable (ce n'est pas pour autant qu'on est capable d'écrire l'algorithme).

Remarque 3.32. *Cette partie du cours est très importante, car on a trouvé un "trou dans le mur des fonctions calculables" et on va étendre le trou (métaphore de M. Deville). C'est-à-dire que maintenant on va utiliser halt pour montrer que d'autres fonctions sont calculables/non calculables. Par exemple par réduction par rapport à halt : si on montre qu'une fonction équivalente à halt est calculable, alors halt est calculable.*

Définition 3.33 (HALT)**.** *HALT est l'ensemble des programmes k et des valeurs x tel que le programme k se termine avec l'entrée x.*

$$HALT = \{(n, x) | P_n(x) \text{se termine}\}$$

Il existe donc au moins un ensemble non récursif K.

Définition 3.34 (K)**.** $K = \begin{array}{l} \{n \mid (n,n) \in HALT\} \\ \{n \mid halt(n,n)=1\} \\ \{n \mid P_n(n) \text{ se termine}\} \end{array}$

K est donc l'ensemble des programmes P_n qui se terminent en recevant comme entrée leur numéro de programme.

Propriété 3.35. *K et HALT ne sont pas récursifs.*

Propriété 3.36. *K et HALT sont récursivement énumérables (car il suffit de lancer le programme et s'il retourne quelque chose il se termine sinon on ne peut rien dire).*

Propriété 3.37. *$\overline{HALT}$ n'est pas récursivement énumérable sinon HALT serait récursif. En effet, on ne peut pas coder un programme qui dirait si un autre programme k, appelé avec la valeur x, ne terminera jamais.*

Propriété 3.38. *$\overline{K}$ n'est pas récursivement énumérable*

3.7 Insuffisance des fonctions totales

Pour l'informaticien, un programme qui ne finit jamais est un vrai problème. Est-ce que le programme va se finir un jour ou est-ce qu'il bouclera à l'infini ? Cette question peut trouver une réponse très rapidement pour des programmes triviaux mais dès que la complexité du programme augmente, il devient très difficile d'y répondre.

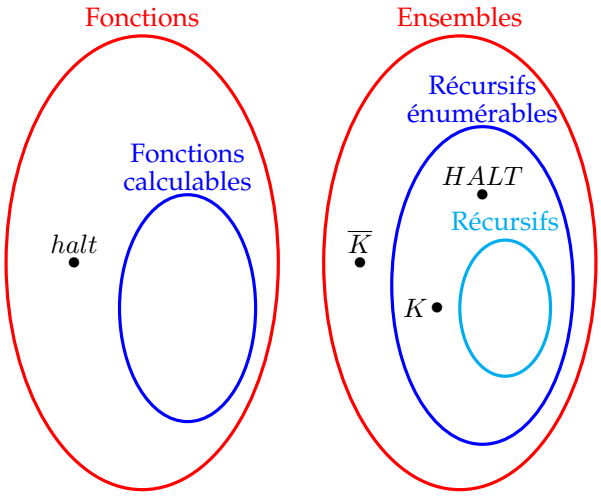


FIGURE 3.1 – Structures de fonctions et ensembles

Pourquoi ne créerions nous pas un langage qui ne peut en aucun cas boucler à l'infini ? Après tout, personne n'a envie d'un programme qui ne s'arrête jamais. Dans ce cas, le modèle de calcul que nous aurons créé aura une fonction halt trivialement calculable (c'est le programme qui retourne 1). Et toutes les fonctions calculées seront des fonctions totales. Cette idée parait plutôt bonne de premier abord mais le bon côté d'un tel langage est accompagné d'un gros inconvénient comme nous allons le voir ci-dessous, il ne permet pas de calculer toutes les fonctions totales calculables.

Exemple 3.39. *Un exemple de modèle de calcul ne permettant que le calcul de fonctions totales est par exemple* ***BLOOP*** *ou encore* ***MiniJava****, c'est-à-dire Java sans récursivité ni boucle while et où les boucles for ne peuvent modifier le compteur de boucle.*

Posons Q un langage (**non trivial**, langage ou les opérations de base tel que l'addition peuvent être effectuées) dont tous les programmes se terminent (ils ne calculent que des fonctions totales) et pour lequel il existe un interpréteur calculable, $interpret(n, x) = \varphi'_n(x)$, où φ'_n est la fonction calculée par le programme Q_n. En partant du fait que tous les programmes se terminent, l'interpréteur est donc une fonction totale puisqu'il donne toujours une réponse.

Théorème 3.40 (Hoare-Allison)**.** *Soit Q un langage de programmation (****non trivial****) dans lequel tous les programmes se terminent. Alors l'interpréteur de Q, la fonction* $\text{interpret}(n, x)$*, n'est pas calculable dans Q.*

Démonstration. Via la diagonalisation de Cantor :
On suppose *interpret* calculable dans Q.

1. On construit une table infinie, où chaque ligne correspond à un programme Q_k de Q :

	0	1	2	...	k	...
Q_0	$interpret(0,0)$	$interpret(0,1)$	$interpret(0,2)$	...	$interpret(0,k)$	...
Q_1	$interpret(1,0)$	$interpret(1,1)$	$interpret(1,2)$	...	$interpret(1,k)$	...
Q_2	$interpret(2,0)$	$interpret(2,1)$	$interpret(2,2)$	...	$interpret(2,k)$	...
:	:	:	:	:	:	:
Q_k	$interpret(k,0)$	$interpret(k,1)$	$interpret(k,2)$	...	$interpret(k,k)$	...
:	:	:	:	:	:	:

2. Sélection de la diagonale $diag(n) = interpret(n, n)$.
3. Modification de cet élément diag pour obtenir $diag'(n) = interpret(n, n) + 1$. $diag'$ est calculable dans Q, car interpret est calculable dans Q (il y a moyen d'écrire un programme qui calcule $diag'$ en utilisant $diag$).

4. Contradiction :
   Comme $diag'$ est calculable dans Q, il existe un programme Q_d qui calcule $diag'$. Mais quelle est la valeur de $diag'(d)$? Par définition de $diag'$, on a $diag'(d) = interpret(d, d) + 1$.
   Mais, comme Q_d calcule $diag'$, par définition $diag'(d) = Q_d(d) = \varphi_d(d) = interpret(d, d)$.
   En effet, calculer $\varphi_d(d)$ revient à interpréter le programme d avec la donnée d.
   On obtient donc $interpret(d, d)+1 = interpret(d, d)$, ce qui est impossible car $interpret(d, d)$ est un entier.
5. Conclusion : $diag'$ n'est pas calculable dans Q, donc $diag$ n'est pas calculable dans Q, et donc $interpret$ n'est pas calculable dans Q.

□

Remarque 3.41. *Si les programmes de Q ne calculaient pas que des fonctions totales, alors ça n'aurait pas posé de problèmes, car si $interpret(d, d) = \perp$, $interpret(d, d) + 1 = \perp$ et il n'y a plus de contradiction. Un programme qui ne se termine pas + quelque chose, reste un programme qui ne se termine pas.*

3.7.1 Implication du théorème 3.40, Hoare Allison

Propriété 3.42. *Si un langage (**non trivial**) ne permet que le calcul de fonctions totales, alors :*

- *l'interpréteur de ce langage n'est pas calculable dans ce langage ;*
- *il existe des fonctions totales non programmables dans ce langage ;*
- *ce langage est* **restrictif.**

Propriété 3.43. *Si on peut programmer l'interpréteur d'un langage L dans L alors il est impossible de programmer la fonction halt dans L car celle-ci n'est pas calculable.*

Propriété 3.44. *Dans un langage de programmation non trivial, il est donc impossible que l'interpréteur et la fonction halt puissent tous deux être programmés.*

Propriété 3.45. *Si on veut pouvoir programmer toutes les fonctions totales dans un langage, le langage doit permettre la programmation de fonctions non totales.*

Propriété 3.46. *L'ensemble $\{n \mid \varphi_n$ est totale$\}$ n'est pas récursif (sinon on saurait créer un langage qui calcule toutes les fonctions totales et uniquement les fonctions totales).*

Théorème 3.47 (fonction universelle)**.** *La fonction universelle (c'est-à-dire l'interpréteur) est $\theta(n, x)$ tel que :*

$$\theta(n, x) = \varphi_n(x)$$

est calculable, c'est à dire qu'il existe z tel que P_z calcule l'interpréteur universel, c'est à dire $\varphi_z = \theta$.

3.8 Extension de fonctions partielles

Théorème 3.48. *Il existe une fonction partielle calculable g telle qu'aucune fonction totale calculable n'est une extension de g.*

Démonstration. Soit $nbStep(n, x)$ = nombre d'instructions exécutées par $\varphi_n(x)$.
Cette fonction vaut $\perp$ si $P_n(x) = \perp$. Il s'agit donc d'une fonction partielle calculable. S'il existe une extension totale calculable de cette fonction, alors la fonction $halt$ serait calculable.
En effet, soit $nbStepPrime(n, x)$ une extension totale calculable de $nbStep$. Pour calculer $halt(n, x)$, il suffit d'exécuter $nbStepPrime(n, x)$ instructions de $P_n(x)$. Si à cette étape $P_n(x)$ s'arrête, c'est que $nbStepPrime(n, x) = nbStep(n, x)$ et que $P_n(x)$ se termine. Si $P_n(x)$ ne s'arrête pas à cette étape, c'est que $nbStep(n, x) = \perp$ et donc que $P_n(x)$ ne se termine pas. On a donc un algorithme pour calculer $halt$, ce qui est impossible. Donc une telle extension totale de $nbStep$ n'existe pas. □

Remarque 3.49. *Ceci implique que si on a une fonction partielle g, il n'est pas toujours possible de créer une fonction totale f qui étend g tel que en dehors du domaine de g, f retourne un code d'erreur. En effet, si cette fonction f existait, halt serait calculable.*

3.9 Théorème de Rice

Il est intéressant de pouvoir analyser un programme afin d'établir ses propriétés, mais est-il possible de déterminer n'importe quelle propriété d'un programme à l'aide d'un algorithme ?

Le théorème de Rice nous répond que **non**, toute propriété sémantique non triviale d'un programme est non calculable par un algorithme.

Voici quelques exemples avant d'attaquer le cas général. Pour chaque exemple, nous pouvons démontrer la non-récursivité de l'ensemble par une réduction à HALT.

Technique de réduction :

1. Nous savons que la fonction halt est non calculable.
2. Nous démontrons que si la fonction f est calculable, alors halt l'est aussi.
3. La fonction f est donc non calculable.

Exemple 3.50. *Soit la fonction :*

$$f(n) = \begin{cases} 1 \text{ si } \varphi_n(x) = \perp \text{ pour tout } x, \\ 0 \text{ sinon.} \end{cases}$$

En supposant que f est calculable, on montre que $halt$ est calculable. Soit le programme F qui calcule la fonction f. Nous construisons $HALT(n, x) \equiv$

— *Construire le programme $P(z) \equiv P_n(x)$*
— *Soit d le numéro de $P(z)$*
— *Si F(d) = 1 alors print(0) sinon print(1)*

Exemple 3.51. *Soit la fonction :*

$$f(n) = \begin{cases} 1 \text{ si } \varphi_n(x) = x \text{ pour tout } x, \\ 0 \text{ sinon.} \end{cases}$$

En supposant que f est calculable, on montre que $halt$ est calculable. Soit le programme F qui calcule la fonction f. Nous construisons $HALT(n, x) \equiv$

— *Construire le programme $P(z) \equiv P_n(x); returnz;$*
— *Soit d le numéro de $P(z)$*
— *Si F(d) = 1 alors print(1) sinon print(0)*

Si $P_n(x)$ s'arrête, $P(z)$ renverra toujours son input donc $F(d)$ renverra 1. Sinon, renverra $F(d)$ renverra 0.

Exemple 3.52. *Soit la fonction :*

$$f(n, m) = \begin{cases} 1 \text{ si } \varphi_n = \varphi_m \text{ pour tout } x, \\ 0 \text{ sinon.} \end{cases}$$

En supposant que f est calculable, on montre que $halt$ est calculable. Soit le programme F qui calcule la fonction f. Nous construisons $HALT(n, x) \equiv$

— *Construire le programme $P_d(z) \equiv P_n(x);$*

— *Construire le programme* $P_{d'}(z) \equiv$ `while true do;`

— *Soit d le numéro de* $P_d(z)$*, et d' le numéro de* $P_{d'}(z)$*.*

— *Si F(d,d') = 1 alors print(0) sinon print(1)*

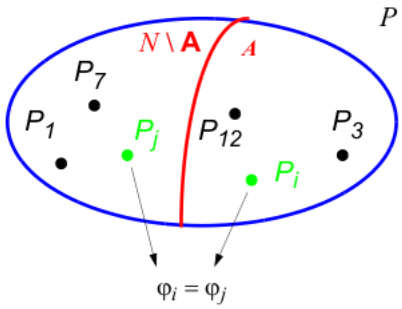
Si $P_n(x)$ *s'arrête,* $F(d, d') = 0$*. Sinon,* $\varphi_d = \varphi_{d'}$ *et donc* $F(d, d') = 1$*. Il suffit de renvoyer* $1 - f(d, d')$ *pour avoir Halt.*

3.9.1 Énoncé et démonstration

Soit un ensemble de programmes A $\subseteq \mathbb{N}$ et $\overline{A} = \mathbb{N} \setminus A$.

Théorème 3.53 (Rice). *Si A est récursif et* $A \neq \emptyset, A \neq \mathbb{N}$*, alors* $\exists i \in A$ *et* $\exists j \in \mathbb{N} \setminus A$ *tel que* $\varphi_i = \varphi_j$*.*

Si A est récursif, qu'il n'est pas vide et qu'il ne contient pas l'ensemble des programmes existant, alors il existe un programme dans A et un programme dans $\overline{A}$ *tel qu'ils calculent la même fonction.*



On utilise le plus souvent le théorème de Rice en ayant recourt à sa contraposée :

Théorème 3.54 (Rice (contraposée)). *Si* $\forall i \in A$ *et* $\forall j \in \overline{A}$ *on a* $\varphi_i \neq \varphi_j$*, alors A est non-récursif ou* $A = \emptyset$ *ou* $A = \mathbb{N}$*.*

Dans de nombreux cas, on peut garantir $A \neq \emptyset$ et $A \neq \mathbb{N}$, ce qui permet de se servir de la contraposée pour démontrer qu'un ensemble est non-récursif sans utiliser la preuve par diagonalisation ou par réduction.

Démonstration. Nous avons que $\forall i \in A$ et $\forall j \in \overline{A}$, $\varphi_i \neq \varphi_j$. Supposons A récursif, $A \neq \emptyset$ et $A \neq \mathbb{N}$.

Dans cette démonstration, nous allons utiliser la méthode de réduction. Sous l'hypothèse que A est récursif, on montre que **HALT** est récursif en construisant un programme qui décide **HALT**. Cela amène alors à une contradiction, car on sait que **HALT** n'est pas récursif. Ce qui implique donc que A ne peut pas être récursif.

1. Construisons le programme suivant P_k :

```
1 while true do;
```

   P_k ne s'arrête jamais, il vaut donc en permanence *bottom* : $\forall x\ \varphi_k(x) = \perp$.
2. $\overline{A} \neq \emptyset$ car $A \neq \mathbb{N}$, supposons $k \in \overline{A}$ (hypothèse sans importance, car montrer que A ou $\overline{A}$ est non récursif revient au même, car A non récursif $\Leftrightarrow$ $\overline{A}$ non récursif).
3. $A \neq \emptyset$ par hypothèse. Prenons un élément quelconque m de cet ensemble ($m \in A$)
4. Par hypothèse $\forall i \in A, \forall j \in \overline{A}\ \varphi_i \neq \varphi_j$. Donc $\varphi_m \neq \varphi_k$.
5. Pour n et x fixé, analysons le programme $P(z)$ suivant. $P(z) \equiv$

```
P_n(x);
P_m(z);
```

Ce programme a un numéro d. Quelle est la fonction $\varphi_d(z)$ calculée par $P(z)$? Il n'y a que deux possibilités.

— Si $P_n(x)$ ne se termine pas, alors le programme $P(z)$ ne se termine pour aucune valeur de z. On a donc $\varphi_d = \varphi_k$.

— Si $P_n(x)$ se termine, alors le programme $P(z)$ calcule la même fonction que $P_m(z)$. On a donc $\varphi_d = \varphi_m$.

Pour déterminer si $\varphi_d = \varphi_k$ ou $\varphi_d = \varphi_m$, il suffit de tester si $d \in A$. Par hypothèse $\forall i \in A, \forall j \in \overline{A}\ \varphi_i \neq \varphi_j$. Nous avons aussi $k \in \overline{A}$ et $m \in A$.

— Si $d \in A$, alors $\varphi_d \neq \varphi_k$, donc $\varphi_d = \varphi_m$.

— Si $d \in \overline{A}$, alors $\varphi_d \neq \varphi_m$, donc $\varphi_d = \varphi_k$.

En résumé : si $d \in A$ alors l'exécution de $P_n(x)$ se termine, sinon l'exécution de $P_n(x)$ ne se termine pas.

On peut alors écrire le programme suivant qui décide **HALT**. $halt(n, x) \equiv$

```
construire un programme P(z)= P_n(x); P_m(z);
d <- numero de P(z);
if d in A then print(1);
else print(0);
```

6. Contradiction, car **HALT** n'est pas récursif.
7. Conclusion : A n'est pas récursif.

□

3.9.2 Analyse

En tenant compte du théorème de Rice, il est maintenant possible de différencier quelles propriétés d'un programme sont déterminables par un algorithme et lesquelles ne le sont pas.

— **Si** la propriété est vérifiée par certains programmes, mais pas tous, et qu'elle est décidable, **alors** il existe deux programmes calculant la même fonction, mais l'un vérifie la propriété, et l'autre pas.

— **Si** une propriété de la fonction calculée par un programme est vérifiée par un programme, mais pas tous, **alors** cette propriété ne peut-être décidée par un algorithme.

— **S**'il existe un algorithme permettant de déterminer si un programme quelconque calcule une fonction ayant cette propriété, **alors** toutes les fonctions calculables ont cette propriété ou aucune fonction calculable n'a cette propriété.

— Aucune question relative aux programmes, vu sous l'angle de la fonction qu'ils calculent, ne peut être décidée par l'application d'un algorithme.

— Les propriétés intéressantes d'un programme concernant la fonction qu'il calcule, non pas sa forme (syntaxe), sont non calculables.

— La plupart des problèmes intéressants au sujet des programmes sont non calculables

3.9.3 Exemples

Pour cela, considérons A comme étant l'ensemble des programmes qui respectent une propriété. Voici quelques exemples et applications du théorème de Rice.

$A_1 =$ $\{i|\varphi_i$ est totale$\}$
P_i s'arrête toujours.

$A_2 =$ $\{i|\varphi_i = f\}$ avec f fixée
P_i calcule la fonction f.

$A_3 =$ $\{i|a \in \text{dom}(\varphi_i)\}$ avec a fixé
$P_i(a)$ se termine.

$A_4 =$ $\{i|\varphi_i(X) = \perp$ pour tout X$\}$
P_i calcule la fonction partielle vide.

$A_5 =$ $\{i|a \in \text{image}(\varphi_i)\}$ avec a fixé
P_i donne au moins une fois le résultat a.

$A_6 =$ $\{i|\,\text{image}(\varphi_i) = \mathbb{N}\}$
P_i donne tous les résultats possibles.

$A_7 =$ $\{i|\varphi_i$est une fonction injective$\}$
P_i calcule une fonction injective.

L'ensemble $A_1, \cdots, A_7$ est non récursif, par la contraposée du théorème de Rice. L'ensemble inverse $\overline{A_1}, \cdots, \overline{A_7}$ n'est par conséquent par récursif non plus.

3.10 Théorème de la paramétrisation

3.10.1 Transformation de programmes

Définition 3.55 (Transformateur de programme)**.** *On peut voir une fonction f : $\mathbb{N} \to \mathbb{N}$ comme une fonction qui prend le numéro d'un programme et qui retourne le numéro d'un autre programme f : $P \to P$. Donc on peut voir P_k, le programme qui calcule f comme un transformateur de programme. En effet, P_k prend un programme en entrée et retourne un programme qui peut être différent.*

Théorème 3.56 (S-m-n)**.** *Pour tout $m, n \geq 0$,*
il existe une fonction totale calculable $S_n^m : \mathbb{N}^{m+1} \to \mathbb{N}$
telle que pour tout k (φ_k est calculable),

$$\varphi_k^{(n+m)}(x_1, ..., x_n, x_{n+1}, ..., x_{n+m}) = \varphi^{(n)}_{S_n^m(k, x_{n+1}, ..., x_{n+m})}(x_1, ..., x_n)$$

Démonstration. Pour prouver que S_n^m est totale calculable, on va montrer comment construire un programme qui calcule $S_n^m(k, x_{n+1}, ..., x_{n+m})$.
Tout d'abord, comme φ_k est calculable, il existe un programme $P_k(x_1, x_2, ..., x_{n+m})$.
On peut donc construire un programme $Q(x_1, ..., x_n)$ qui calcule $P_k(x_1, x_2, ..., x_{n+m})$.
$x_1, ..., x_n$ restent des arguments du programme, tandis que $x_{n+1}, ..., x_{n+m}$ deviennent des valeurs fixées.
Notre programme qui calcule $S_n^m(k, x_{n+1}, ..., x_{n+m})$ n'a plus qu'à retourner le numéro du programme Q.
En résumé, $S_n^m(k, x_{n+1}, ..., x_{n+m}) \equiv$

```
construire Q(x1,x2,...,xn) = P_k(x1,x2,...xn,...,xn+m);
d <- numero du programme Q;
print(d);
```

□

Théorème 3.57 (S). *C'est la propriété S-m-n affaiblie.*

$$\forall\, k\ \exists\, S \text{ (totale calculable)} : \varphi_k(x,y) = \varphi_{S(y)}(x)$$

Démonstration. Par le théorème 3.56 (S-m-n),

$$\exists\, S \text{ (totale calculable)}\ \forall\, k : \varphi_k(x,y) = \varphi_{S(k,y)}(x)$$

$$\Rightarrow \forall\, k\ \exists\, S \ \text{ (totale calculable) } : \varphi_k(x,y) = \varphi_{S(k,y)}(x)$$

Pour un k donné, on note $S_k(y) = S(k,y)$. S_k est calculable puisque S l'est. On a bien $\varphi_k(x,y) = \varphi_{S_k(y)}(x)$. □

Remarque 3.58. *On peut voir le théorème 3.56, S-m-n, comme :*
Étant donné $m, n \geq 0$, il existe un transformateur de programme, S_n^m, qui recevant comme données un programme P_k à $n+m$ arguments et m valeurs $v_1, ..., v_m$ fournit comme résultat un programme P à n arguments tel que $P(x_1, ..., x_n)$ calcule la même fonction que $P_k(x_1, ..., x_n, , v_1, ..., v_m)$ (ce programme effectue une projection)

Remarque 3.59. *On peut donc voir la transformation de programmes S_n^m comme un programme qui particularise un autre programme à $m+n$ argument en rendant constant les m derniers paramètres.*

3.11 Théorème du point fixe

Théorème 3.60 (Point fixe). *Soient $n \geq 0$ et f fonction totale calculable, il existe k tel que $\varphi_k^{(n)} = \varphi_{f(k)}^{(n)}$.*

Remarque 3.61. *Le théorème 3.60 n'est pas très intuitif. Mais on peut le voir comme : quel que soit un transformateur de programme T qui calcule f (n'importe quelle fonction totale calculable peut être vue comme un transformateur de programme), il existe deux programmes P_k et P_j tels que*

- *P_j est la transformation de P_k via T,*
- *P_k et P_j calcule la **même** fonction.*

Démonstration. Pour commencer la démonstration, on va sortir 3 "lapins" d'un chapeau (analogie aux tours de magie) :

1^{er} **lapin :** Soit $h(u,v) = \begin{cases} \varphi_{\varphi_u(u)}(v) & \text{si } \varphi_u(u) \neq \perp, \\ \perp & \text{sinon.} \end{cases}$

h est calculable

Remarque 3.62. *on peut créer un programme qui calcule h :*
Input : u, v
$a = P_z(u,u)$
$P_z(a,v)$
où $\varphi_z = \theta$ est la fonction universelle.

2^{eme} **lapin :** $h(u,v) = \varphi_{S(u)}(v)$
Ceci est une application de la propriété S (propriété S-m-n affaiblie), avec S totale calculable.

3^{eme} **lapin :** Soit $g(u) = f(S(u))$
g est totale calculable car S et f le sont (f est la fonction donnée dans le théorème).

$$\exists k' \text{ tel que } \varphi_{k'}(u) = g(u) = f(S(u))$$

On a que k' est une constante et par le lapin 2 :

$$h(k', v) = \varphi_{S(k')}(v)$$

Par le lapin 1 et car $g = \varphi_{k'}$ est une fonction totale calculable, on a :

$$h(k', v) = \varphi_{\varphi_{k'}(k')}(v)$$

Par le lapin 3, on a que $\varphi_{k'}(u) = g(u) = f(S(u))$ donc :

$$h(k', v) = \varphi_{f(S(k'))}(v)$$

Par le lapin 2, on a :

$$\varphi_{S(k')}(v) = \varphi_{f(S(k'))}(v)$$

Si on pose que $S(k') = k$ on a bien

$$\varphi_k(v) = \varphi_{f(k)}(v)$$

Ce qui conclut notre démonstration.

Remarque 3.63. *La fonction $S(k')$ pourrait produire le programme suivant :*

Input : v
$a = P_z(k', k')$
$P_z(a, v)$

On remarque que comme S et f sont totales, g l'est aussi donc $P_z(k', k')$ se terminera toujours et donne pour résultat $f(S(k'))$ (lapin 3).

Que se passe-t-il quand on exécute le programme généré par $S(k')$ sur une donnée v quelconque ? La seconde instruction se termine et on obtient $a = f(S(k'))$. Ensuite on exécute le programme $P_z(a, v) = P_a(v)$ qui fournit le résultat final.
Exécuter le programme généré par $S(k')$ sur une donnée v revient donc à exécuter le programme $f(S(k'))$ sur cette donnée v. On a donc

$$\varphi_{S(k')} = \varphi_{f(S(k'))}.$$

Le programme $S(k')$ est bien un point fixe de f.

□

Remarque 3.64 (Conséquences Point fixe)**.** *Grâce à ce théorème qui se base sur l'unique propriété de S, nous pouvons démontrer plein de choses comme :*

- *HALT non-calculable*
- *Le théorème de Rice*
- *Le théorème de Hoare-Allison via la démonstration de HALT non-calculable*

Le théorème du point fixe c'est la base de la calculabilité. Il fixe toutes les limites de la calculabilité.

Remarque 3.65 (Démonstration du théorème de Rice grâce au point fixe)**.** *À l'aide du point fixe, on peut démontrer plus simplement le théorème de Rice :*

Si A est récursif et $A \neq \emptyset \neq \bar{A}$, alors il existe $i \in A$ et $j \in \bar{A}$ tel que $\varphi_i = \varphi_j$.

Démonstration. Soit un ensemble A tel que $A \neq \emptyset$ et $\bar{A} \neq \mathbb{N}$. Il suffit de montrer que si $\forall i \in A, \forall j \in \bar{A} : \varphi_i \neq \varphi_j$ (1) alors A est non récursif.
En supposant A récursif, on arrive à une contradiction. En effet, soit $n \in A$ et $m \in \bar{A}$, définissons la fonction

$$f(x) = \begin{cases} m & \text{si } x \in A, \\ n & \text{si } x \in \bar{A}. \end{cases}$$

Comme A est récursif, la fonction f est total calculable. Dès lors, par le théorème du point fixe, il existe k tel que

$$\varphi_{f(k)} = \varphi_k$$

La valeur k est-elle dans A ou dans $\bar{A}$?

— Si $k \in A$, alors $f(k) = m$. On a donc $\varphi_m = \varphi_k$, ce qui est contradictoire avec (1) car $m \in \bar{A}$.

— Si $k \in \bar{A}$, alors $f(k) = n$. On a donc $\varphi_n = \varphi_k$, ce qui est contradictoire avec (1) car $n \in A$.

□

Remarque 3.66 (Démonstration de K grâce au point fixe)**.** *On peut démontrer que K n'est pas récursif.*

Démonstration. Soit n un programme qui ne s'arrête jamais, et m un programme qui calcule la fonction identité :

— $\varphi_n(x) = \perp \quad \forall\, x$

— $\varphi_m(x) = x \quad \forall\, x$

Posons maintenant la fonction suivante :

— $f(x) = \begin{cases} n & \text{si } x \in \text{K} \\ m & \text{si } x \notin \text{K} \end{cases}$

On montre que par construction pour tout k, $\varphi_{f(k)} \neq \varphi_k$.

— Si $k \in K$, $\varphi_k(k) \neq \perp$, $f(k) = n$ et $\varphi_{f(k)}(k) = \varphi_n(k) = \perp \neq \varphi_k(k)$ donc $\varphi_{f(k)} \neq \varphi_k$.

— Si $k \notin K$, $\varphi_k(k) = \perp$, $f(k) = m$ et $\varphi_{f(k)}(k) = \varphi_m(k) \neq \perp = \varphi_k(k)$ donc $\varphi_{f(k)} \neq \varphi_k$.

Si K est récursif, alors f est total calculable. Par le théorème du point fixe, il existe k tel que $\varphi_k = \varphi_{f(k)}$. Ce qui est contradictoire.

K n'est donc pas récursif. □

3.12 Autres problèmes non calculables

Définition 3.67 (Problème de correspondance de Post)**.** *Soit deux listes U et V de mots non vides sur un alphabet Σ :*

— $U = u_1, u_2, ..., u_k$

— $V = v_1, v_2, ..., v_k$

*Le problème consiste à décider s'il existe une suite d'entiers $i_1, i_2, .., i_n$ telle que les mots $u_{i_1}, u_{i_2}, ..., u_{i_n}$ et $v_{i_1}, v_{i_2}, ..., v_{i_n}$ soient **identiques**.*
Pour démontrer l'indécidabilité de ce problème, Post à introduit un nouveau modèle, la machine de Post qui ressemble à une machine de Turing.

Définition 3.68 (Problème des équations diophantiennes)**.** *Décider si une équation polynomiale de degré supérieur ou égal à 4 possède une solution entière.*

C'est le 10e problème de Hilbert, résolu en 1970 par Matiyasevich ; voir [**?**] pour une description de la solution et des remarques historiques. Ce résultat implique, en particulier, qu'il n'existe pas d'algorithme pour résoudre la programmation quadratique entière ; voir [**?**]. Un bon aperçu des questions de non décidabilité pour les équation polynomiale est [**?**].

Il n'existe pas d'algorithme résolvant ces problèmes. Il existe beaucoup d'autres problèmes non calculables. Il y a d'autres exemples sur les grammaires dans le cours.

3.13 Nombres calculables

Définition 3.69 (Nombre réel)**.** *Un nombre réel est défini comme la limite d'une suite (convergente) de nombres rationnels :* $\lim_{n\to+\infty} |x - s(n)| = 0$ *ou* s *est une fonction totale.*

Définition 3.70 (Nombre réel calculable)**.** *Un nombre réel x est calculable s'il existe une fonction totale calculable* s *tel que pour tout* n *:* $|x - s(n)| \leq 2^{-n}$*.*

Remarque 3.71. *Donc un nombre est calculable s'il existe un programme qui peut l'approximer aussi près que l'on veut. Par exemple* π *et* e *sont calculables*

Propriété 3.72. *L'ensemble des nombres réels calculables est énumérable, car on peut énumérer les fonctions totales calculables.*

Propriété 3.73. *Il existe des nombres réels non calculables.*

Propriété 3.74. *Il existe des nombres réels non calculables qui peuvent être définis de manière finie.* [1]

3.14 Conclusion

[Conclusion très partielle...]
Le théorème S-m-n permet de démontrer le théorème du point fixe. Le théorème du point fixe est un résultat central de la calculabilité. Il implique le théorème de Rice, la non-récursivité de K et la non-calculabilité de la fonction halt.

1. Par exemple l'Oméga de Chaitin, qui est défini comme étant la probabilité qu'un programme auto-délimité, généré aléatoirement, finisse par s'arrêter.

Chapitre 4

Modèles de calculabilité

4.1 Familles de modèles

Il y a deux grandes familles de modèles :

— Modèle de calcul (calcule une réponse)
— Modèle de langages (décide l'appartenance à un ensemble)

4.1.1 Modèles de calcul

L'objectif est de modéliser le concept de fonctions calculables, processus de calcul, algorithme effectif.

On peut encore classer les modèles de calcul en 2 catégories, les modèles déterministes et les modèles non déterministes.

Définition 4.1 (Modèles déterministes)**.** *une seule exécution possible*

Définition 4.2 (Modèles non déterministes)**.** *il existe plusieurs exécutions possibles*

On va voir les modèles de calcul suivant :

— Automate fini
— Automate à pile
— Machine de Turing
— Langages de programmation
— Lambda calcul
— Fonction récursive

Mais il en existe beaucoup d'autres.

4.1.2 Modèles de langage

Un langage est défini par une grammaire formelle. L'objectif est de modéliser une classe de langages. Le langage est alors soit un ensemble récursif, soit un ensemble récursivement énumérable.

4.2 Langages de programmation

C'est un modèle possible de la calculabilité. Pour définir un langage de programmation comme modèle de la calculabilité, il faut définir :

— Syntaxe du langage
— Sémantique du langage
— Convention de représentation d'une fonction par un programme

On se pose la question de savoir s'il y a des langages plus puissants que d'autres. On va montrer que tous les langages complets sont équivalents. (Et la plupart des langages sont complets.)

Mais, il existe aussi des langages qui ne sont pas complets comme le langage BLOOP (bounded loop).

Définition 4.3. *BLOOP : Sous ensemble de Java qui ne calcule que des fonctions totales. (pas de boucle while, boucle for mais sans modification du compteur dans le for, pas de goto en arrière, pas de fonctions récursives)*

BLOOP a donc toutes les propriétés qui découlent du chapitre précédent :

Propriété 4.4. *Tous les programmes BLOOP se terminent :*
⇒ BLOOP ne calcule que des fonctions totales
⇒ L'interpréteur est une fonction totale non calculable en BLOOP (Hoare-Allison)
⇒ BLOOP ne calcule pas toutes les fonctions totales
⇒ BLOOP n'est pas un modèle complet de la calculabilité

Cependant il existe un compilateur des programmes BLOOP (Java).

4.2.1 Langage de programmation non déterministe

Remarque 4.5. *Il est difficile d'avoir de l'intuition sur cette partie. On peut voir un programme ND comme un programme qui produit des résultats différents d'une exécution à l'autre. On peut représenter toutes les exécutions possibles du programmes sous forme de branches d'un arbre. Pour analyser la complexité, on ne considère que la profondeur de l'arbre (longueur de la branche la plus longue).*

On va un introduire un nouveau langage ND-Java qui est le langage Java auquel on ajoute le non-déterminisme sous la forme d'une fonction prédéfinie $choose(n)$. Celle-ci retourne un entier compris entre 0 et n et elle est non déterministe.

On peut voir un programme ND de 2 manières différentes :

1. Il calcule une relation plutôt qu'une fonction
2. C'est un moyen de décider si un élément appartient à un ensemble

On considère l'approche 2 en calculabilité.

Définition 4.6 (ND-récursif)**.** *Un ensemble $A \subseteq \mathbb{N}$ est ND-récursif s'il existe un ND-programme tel que lorsqu'il reçoit comme donnée n'importe quel nombre naturel x*
si $x \in A$ alors il existe une exécution fournissant tôt ou tard comme résultat 1
si $x \notin A$ alors toutes les exécutions fournissent tôt ou tard comme résultat 0

Définition 4.7 (ND-récursivement énumérable)**.** *Un ensemble $A \subseteq \mathbb{N}$ est ND-récursivement énumérable s'il existe un ND-programme tel que lorsqu'il reçoit comme donnée n'importe quel nombre naturel x*
si $x \in A$ alors il existe une exécution fournissant tôt ou tard comme résultat 1
si $x \notin A$ alors les exécutions possibles ne se terminent pas ou retournent un résultat $\neq 1$

Propriété 4.8. *On peut simuler les exécutions d'un ND-programme à l'aide d'un programme déterministe.*

Démonstration. Soit un programme non-déterministe quelconque à N branches (N *threads*). Montrons qu'on peut construire un programme déterministe équivalent. Il suffit de créer un programme qui exécute séquentiellement chacune des N branches de l'arbre. Cependant les branches pouvant être infinies, il est nécessaire d'explorer ces branches de l' arbre d'exécution en largeur d'abord (*Breadth First Search*). La complexité temporelle du programme ainsi construit devient exponentielle. En effet, un arbre de profondeur k a un nombre exponentiel de noeuds. □

Exemple : Le ND-programme à gauche (figure 4.1) comporte 3 branches et a une hauteur de 3. Il est tranformé en un programme D, à droite (fiure 4.2) qui a une longueur de 9.

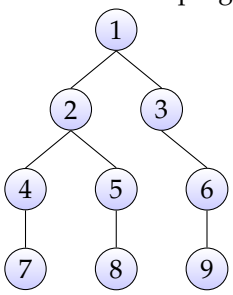


FIGURE 4.1 – ND-programme

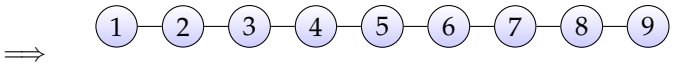


FIGURE 4.2 – D-programme

Propriété 4.9. *Un ensemble est ND-récursif ssi il est récursif*

Démonstration. Si un ensemble A quelconque est ND-récursif, alors il est récursif. En effet, s'il est ND-récursif, il existe un ND-programme qui permet de décider l'ensemble A (voir définition 4.6). Par la propriété 4.8, on sait qu'il existe alors un programme déterministe équivalent qui décide A. À ce programme correspond un algorithme. Selon la définition 3.6, l'ensemble A est alors récursif.

Inversement, si un ensemble A quelconque est récursif, alors il est ND-récursif. En effet, par définition, il existe un programme D qui décide A. Ce programme peut être vu comme un programme ND avec seulement une branche. A est donc également ND-récursif. □

Propriété 4.10. *Un ensemble est ND-récursivement énumérable ssi il est récursivement énumérable*

Démonstration. Si un ensemble A quelconque est ND-récursivement énumérable, alors il est récursivement énumérable. En effet, s'il est ND-récursivement énumérable, il existe un ND-programme dont le comportement est décrit à la définition 4.7. Par la propriété 4.8, on sait qu'il existe alors un programme déterministe qui se comporte de manière équivalente. À ce programme correspond un algorithme. Selon la définition 3.7, l'ensemble A est alors récursivement énumérable.

Inversement, si un ensemble A quelconque est récursivement énumérable, alors il est ND-récursivement énumérable. En effet, par définition, il existe un programme D qui se comporte comme décrit dans la définition 3.7. Ce programme peut être vu comme un programme ND avec seulement une branche. A est donc également ND-récursivement énumérable. □

4.3 Automates finis FA

Objectif : Décider si un mot donné appartient ou non à un langage.

Utilisation : Utilisé dans les interfaces pour les humains (par exemple, les distributeurs).

4.3.1 Modèles des automates finis

Un automate fini est composé de :

— Σ : ensemble fini de symboles
— S : ensemble fini d'états
— $s_0 \in S$: état initial
— $A \subseteq S$: ensemble des états acceptants
— $\delta : S \times \Sigma \to S$: fonction de transition

Fonctionnement

— départ avec un état initial
— parcours des symboles du mot d'entrée, un à un
— à chaque symbole lu, l'état change (fonction de transition δ) en fonction de l'état courant et du symbole lu
— l'état final est l'état après avoir parcouru tous les symboles en entrée
— l'état final peut-être acceptant ou non

Remarque 4.11. *Il n'y a donc pas de mémoire. De plus, un automate peut-être simulé par un programme Java.*

Propriété 4.12. *Un automate fini définit un ensemble récursif de mots* $= \{m \mid m$ *est accepté par FA*$\}$

Propriété 4.13. *Certains ensembles récursifs ne peuvent pas être reconnus par un automate fini. Par exemple* $L = \{a^n b^n \mid n \geq 0\}$ *(parce qu'il nécessiterait un nombre infini d'états)*

Propriété 4.14. *L'interpréteur des automates finis est calculable, mais ne peut pas être représenté par un automate fini, car ce n'est pas un* ***modèle complet*** *de la calculabilité (Hoare-Allison)*

Définition 4.15 (Langage régulier)**.** *Un langage régulier est un langage défini par une expression régulière.*

Exemple Un exemple simple de mécanisme qui peut être modélisé par un automate fini est le portillon d'accès. Dans le métro par exemple, lorsqu'un ticket est inséré, le portillon se déverrouille et le passage d'un usagé est autorisé, lorsque celui ci est passé, il se verrouille à nouveau. Le portillon peut être modélisé comme un automate fini comportant deux états : verrouillés et déverrouillé. L'état peut-être modifié par l'ajout d'un ticket/jeton (entrée : jeton). Soit lorsque l'utilisateur pousse les barres du portillon (entrée : pousser). La fonction de transition est représentée par la table 4.1. La figure 4.3 illustre le diagramme d'état de cet automate.

	Pousser	Jeton
Verrouillé	Verrouillé	Déverrouillé
Déverrouillé	Verrouillé	Déverrouillé

TABLE 4.1 – Fonction de transition du portillon

Un exemple plus complexe est le célèbre problème du passeur. Celui ci doit traverser une rivière, avec un loup, une salade et une chèvre. Dans sa barque il ne peut prendre qu'un objet avec lui. La difficulté supplémentaire est que la chèvre et le loup ne peuvent pas rester ensemble et la chèvre ne peut pas rester seule avec la salade. Chaque état représente les objets se trouvant sur l'autre rive, P étant le passeur, C la chèvre, L le loup et S la salade. Sur les flèches, la lettre correspond à l'objet transporté avec lui lors de la traversée. Au début rien n'a été transporté, à la

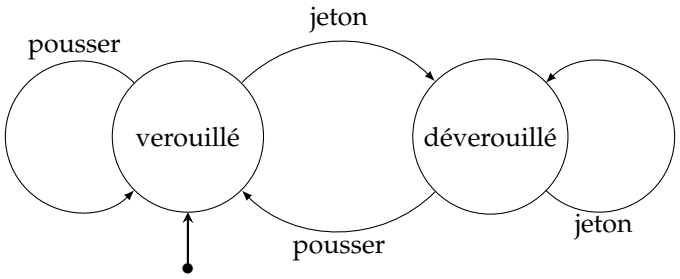


FIGURE 4.3 – Diagramme d'état du portillon
Par ManiacParisien — Travail personnel, CC BY-SA 4.0, https ://commons.wikimedia.org/w/index.php ?curid=47664521

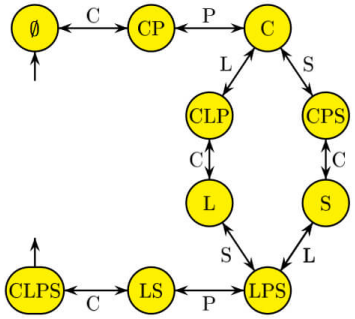


FIGURE 4.4 – Diagramme d'état du passeur
Par ManiacParisien — Travail personnel, CC BY-SA 4.0, https ://commons.wikimedia.org/w/index.php ?curid=47815891

fin, les trois objets et le passeur se retrouvent sur l'autre rive. La figure 4.4 illustre le diagramme d'état de cette énigme.

Un autre exemple d'automate fini est un distributeur de boissons qui reçoit une pièce d'argent et un choix de boisson puis sert la boisson choisie. Par exemple, nous pouvons considérer que la machine reçoit une pièce de 2 euros et a uniquement trois choix Fanta, Coca, Eau. Un tel système peut être modélisé comme suit :

Les transitions avec les mêmes états de départs et d'arrivées et la même sortie ont une seule flèche étiquetée avec l'entrée / la sortie.

Propriétés

Propriété 4.16. *Un automate fini définit un ensemble récursif de mots* $= \{m \mid m$ *est accepté par FA*$\}$.

Propriété 4.17. *Certains ensembles récursifs ne peuvent pas être reconnus par un automate fini.*

Démonstration. Par exemple $L = \{a^n b^n \mid n \geq 0\}$: l'automate doit compter le nombre de a pour vérifier qu'il y a autant de b. Mais il n'a pas de mémoire, donc pas de moyen de retenir le nombre n.

Par contre, un automate fini pourrait reconnaître l'ensemble $L' = \{a^n b^m \mid n, m \geq 0\}$. □

Propriété 4.18. *L'interpréteur des automates finis est calculable, mais ne peut pas être représenté par un automate fini, car ce n'est pas un* ***modèle complet*** *de la calculabilité (Hoare-Allison).*

Définition 4.19 (Langage régulier)**.** *est un langage défini par une expression régulière.*

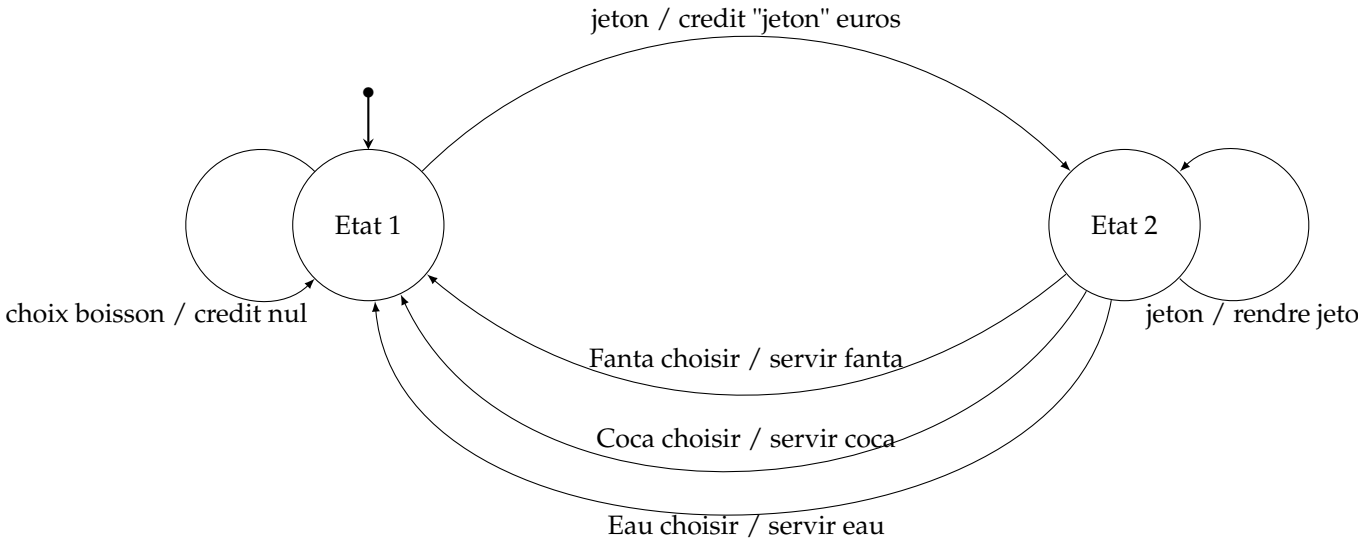


FIGURE 4.5 – Diagramme d'état d'un distributeur de boisson simple
Inspirer du TD7 : Automates Par Stephane Devismes Université Grenoble Alpes

Définition 4.20 (Expression régulière)**.** *Dans le cours, la syntaxe d'une expression régulière est la suivante :*

+ *ou*

. *concaténation*

***** *fermeture de Kleene* [1]

( ) *répétition*

4.3.2 Extension des automates finis

NDFA On étend le modèle en permettant d'avoir plusieurs transitions possibles pour une paire <état,symbole>. Ce qui implique que plusieurs exécutions sont possibles. On a donc plus une fonction de transition mais une relation de transition.

De même, pour un ND programme, un mot est accepté par un NDFA s'il existe au moins une exécution où l'état final est acceptant. Dans l'autre sens, un mot n'est pas accepté si aucune exécution ne se termine avec l'état final acceptant.

Propriété 4.21. *Si un ensemble récursif est défini par un NDFA, alors cet ensemble est défini par un FA.*

Propriété 4.22. *Un NDFA définit un ensemble récursif de mots.*

Ajout de transitions vides ϵ On peut encore étendre le modèle NDFA en rajoutant une possibilité de transition sans lire de symbole (transition spontanée). Ça a la même puissance et les mêmes propriétés qu'un NDFA.

1. définit un groupe qui existe zéro, une ou plusieurs fois

4.4 Automate à pile PDA

C'est une extension du modèle des automates finis. On ajoute une mémoire avec la pile de symboles. Les différences principales sont :

— la transition entre états dépend du symbole lu et du symbole au sommet de la pile

— chaque transition peut enlever le sommet de la pile et empiler de nouveaux éléments ou ne pas changer la pile.

Objectif : Décider si le mot donné appartient ou non à un langage.

Utilisation : Utilisé dans les compilateurs.

Composition : On rajoute Γ et on change la fonction de transition en une nouvelle relation de transition.

— Σ : ensemble fini de symboles d'entrée

— Γ : ensemble fini de symboles de pile

— S : ensemble fini d'états

— $s_0 \in S$: état initial

— $A \subseteq S$: ensemble des états acceptants

— $\Delta \subset S \times \Sigma \times \Gamma \to S \times \Gamma^*$: relation de transition (finie)

Propriété 4.23. *Tout comme un NDFA, un PDA définit un ensemble récursif de mots (langage récursif).*

Convention :

— Z est le symbole initial de la pile (pile vide)

— ϵ signifie qu'aucun symbole ne doit être lu pour cette transition (symbole "vide")

— A, B / C : A est le symbole lu, B est le symbole au sommet de la pile et C est ce qui va remplacer le sommet de la pile (peut-être un xB pour rajouter x sur la pile, ϵ pour retirer B du sommet de la pile, ou juste B pour ne pas changer le sommet)

Propriété 4.24. *Certains ensembles récursifs ne peuvent pas être reconnus par un automate à pile.*

Démonstration. Par exemple $\{a^n b^n a^n | n \geq 1\}$: notre automate doit d'abord retenir le nombre n dans la première série de a, puis vérifier qu'il y a bien n occurrences de b puis n occurrences de a dans la deuxième série. Le seul moyen de retenir n est d'empiler un symbole dans la pile chaque fois qu'on lit un a, de sorte à avoir n symboles empilés après la première série de a.

Ensuite, on enlève un symbole chaque fois qu'on lit un b, sachant qu'on doit arriver au bout de la série de b en même temps qu'on arrive au fond de la pile. Mais impossible alors de savoir combien d'occurrence de a il faut compter dans la deuxième série de a, on ne connaît plus le nombre n ! □

Propriété 4.25. *Les automates à pile sont plus puissants que les automates finis (ils peuvent reconnaitre plus d'ensembles)*

Propriété 4.26. *Ce n'est pas un modèle complet de la calculabilité donc par Hoare- Allison, l'interpréteur n'est pas calculable dans le modèle.*

4.5 Grammaires et modèles de calcul

Objectif : Définition d'un langage (ensemble de mots) et à partir de la grammaire on peut générer/dériver les mots du langage.

Utilisation : Utilisé pour la définition de langages de programmation, pour l'analyse du langage naturel...

Composition du modèle :

- Σ : alphabet
- les éléments de Σ sont des symboles terminaux
- autres symboles utilisés durant la dérivation : symboles non terminaux (A,B, ..., <dig>,..)
- S : point de départ de la dérivation (symbole non terminal)

Définition 4.27 (Règle de production)**.** *On appelle un ensemble de règles de dérivation des règles de production.*

Exemple 4.28.
- $\Sigma = 0, 1, 2$
- $S \rightarrow < Dig >$
- $< Dig > \rightarrow D$
- $D \rightarrow 0|1|2|\epsilon$ *(ϵ signifie rien)*

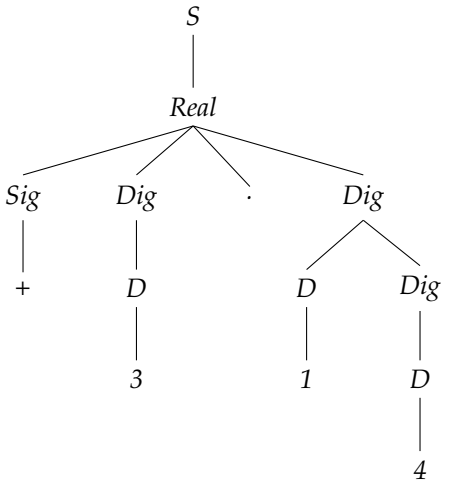
Exemple 4.29 (Grammaire des réels dans Java)**.** $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ., E\}$ *avec les règles de production suivantes :*

- $S \rightarrow < Real >$
- $< Real > \rightarrow < Sig >< Dig > . < Dig >$
- $< Real > \rightarrow < Sig >< Dig > . < Dig > E < Exp >$
- $< Real > \rightarrow < Sig >< Dig > E < Exp >$
- $< Sig > \rightarrow \epsilon \mid + \mid -$
- $< Dig > \rightarrow D$
- $< Dig > \rightarrow D < Dig >$
- $D \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \mid$
- $< Exp > \rightarrow < Dig >$

Par exemple, le réel $+3.14$ est dérivé comme suit (leftmost derivation) :

$$\begin{aligned} S &\rightarrow < Real > \\ &\rightarrow < Sig >< Dig > . < Dig > \\ &\rightarrow + < Dig > . < Dig > \\ &\rightarrow +D. < Dig > \\ &\rightarrow +3. < Dig > \\ &\rightarrow +3.D < Dig > \\ &\rightarrow +3.1 < Dig > \\ &\rightarrow +3.1D \\ &\rightarrow +3.14 \end{aligned}$$

Ce même exemple peut être représenté par un arbre syntaxique :



Définition 4.30 (Dériver)**.** *Appliquer des règles de la grammaire pour vérifier si une chaîne de symbole appartient au langage (on part d'une chaîne de symboles et on vérifie les règles sur celle-ci).*

Définition 4.31 (Inférer)**.** *Dérivation dans "le sens contraire", c'est-à-dire, on part des règles de grammaire et on génère une chaîne de symboles.*

Définition 4.32 (Arbre syntaxique)**.** *Un arbre syntaxique permet de représenter la dérivation, chaque noeud correspond à un symbole terminal ou non. Les arêtes correspondent à l'application d'une règle. Il y a plusieurs nœuds enfants si la règle "génère" plusieurs symboles.*

Propriété 4.33. *On peut dériver de plusieurs façons équivalentes, leftmost (on dérive toujours le plus à gauche d'abord), rightmost (contraire de leftmost) ou aucun des deux.*

4.5.1 Hiérarchie de Chomsky

Chomsky a défini 4 types de grammaires formelles. On peut les classer selon leur "puissance".

Définition 4.34 (Puissance d'une grammaire)**.** *Une grammaire A est plus puissante qu'une grammaire B si on peut définir plus de langages avec A qu'avec B.*

On peut aussi faire correspondre chaque type de grammaire avec un type de calcul permettant de reconnaitre un langage de cette grammaire.

Type	Type de grammaire	Modèle de calcul
3	régulière	Automate fini
2	hors contexte	Automate à pile
1	sensible au contexte	Machine de Turing à ruban fini
0	récursivement énumérable	Machine de Turing

Chaque type de grammaire est défini par une règle de production A $\rightarrow$ B. Il y a des conditions différentes sur A et B selon le type de grammaire.

4.5.2 Grammaires régulières

Règle de production :

— $A \rightarrow \omega B$

— $A \rightarrow \omega$

Conditions :

— $\omega \in \Sigma^*$, c'est-à-dire ω est une chaîne de symboles terminaux.
— A et B sont des symboles non terminaux.

Exemple 4.35. *Règles de dérivation :*

— $S \to abS$
— $S \to \epsilon$

Cette grammaire définit le langage $L1 = \{(ab)^n \mid n \geq 0\}$. *Ce langage peut-être aussi défini par une expression régulière :* $L1 = (ab)^*$.

4.5.3 Grammaires hors contexte

Cette grammaire est importante, car il suffit de lui rajouter la portée des variables pour définir la syntaxe d'un langage de programmation.

Règle de production :

— $A \to \beta$

Conditions :

— β est une chaîne de symboles composée de symboles terminaux ou non
— A est un symbole non terminal

Exemple 4.36. *Règle de dérivation :*

— $S \to aSb$
— $S \to \epsilon$

Le langage défini par cette grammaire est $L1 = \{a^n b^n | n \geq 0\}$.

4.5.4 Grammaires sensibles au contexte

Règle de production :

— $\alpha \to \beta$

Conditions :

— α et β sont des chaînes de symboles composées de symboles terminaux ou non.
— β contient au moins autant de symboles que α.

Exemple 4.37. *Règles de dérivation :*

— $S \to aSBA$
— $S \to abA$
— $AB \to BA$

— $bB \rightarrow bb$

— $bA \rightarrow ba$

— $aA \rightarrow aa$

Le langage défini par cette grammaire est $L1 = \{a^n b^n a^n | n \geq 0\}$.

4.5.5 Grammaires sans restriction

Règle de production :

— $\alpha \rightarrow \beta$

Conditions :

— α et β sont des chaînes de symboles composées de symboles terminaux ou non.

Exemple 4.38. *Il y a donc moyen de créer des règles qui bouclent :*
$\alpha \rightarrow \beta$
$\beta \rightarrow \alpha$

4.6 Machines de Turing

Intérêt : Le modèle des machines de Turing est le modèle le plus simple, le plus élémentaire et le plus puissant possible (c'est un modèle complet de la calculabilité). Il permet une définition précise de procédures, d'algorithmes ou encore de calculs.

Composition "abstraite" :

Ruban Suite de cases potentiellement infinie (des 2 côtés), mais à chaque moment, le ruban nécessaire est fini

Tête Une seule tête, sur une case qui peut écrire et lire la case sur laquelle elle est

Contrôle Dirige les actions/opérations

4.6.1 Contrôle

Le contrôleur est composé d'un nombre d'états fini dont un état initial et un final. Il contient un programme (des instructions).

Définition 4.39 (Une instruction)**.** *Une instruction a la forme*

$$< q, c > \quad \rightarrow \quad < new_q, Mouv, new_c >$$

— q *: état courant*

— c *: symbole sous la tête de lecture*

— new_c *: symbole à écrire sous la tête de lecture*

— $Mouv$ *: G ou D, mouvement que la tête de lecture doit faire*

— new_q *: le nouvel état*

4.6.2 Modélisation

Pour définir une machine de Turing, il faut :

— Σ : ensemble fini de symboles d'entrée
— Γ : ensemble fini de symboles de ruban
— S : ensemble fini d'états
— $s_0 \in S$: état initial
— $stop \in S$: état d'arrêt
— $\delta : S \times \Gamma \to S \times \{G, D\} \times \Gamma$: fonction de transition (finie)

Il faut aussi que $\Sigma \subset \Gamma$ et que B $\in \Gamma$ mais que B $\notin \Sigma$

Définition 4.40. *B correspond au symbole blanc.*

4.6.3 Exécution

Au départ il y a juste les données d'entrée sur le ruban. Sur les autres cases, il y a le symbole B. La tête de lecture se trouve sur la première case des données. Tant que c'est possible, on applique des instructions. Il y a 2 cas possibles pour l'arrêt : soit l'état devient stop, soit il n'y a plus d'instruction applicable.

Le résultat est le contenu du ruban à l'état stop. Si la machine ne s'arrête pas sur l'état stop alors il n'y a pas de résultat.

Exemple 4.41. *Étant donné qu'une machine de Turing peut calculer une fonction, il existe un nombre important de machine de Turing. Celles-ci peuvent avoir des fonctions allant du 'très simple' au 'très complexe'. Par exemple une machine de Turing peut déterminer si un nombre est pair ou impair (en regardant si le dernier bit est égal à zéro ou à un), vérifier si le nombre est un multiple de 42, multiplier un chiffre par deux (il suffit de positionner la tête de lecture à droite et d'ajouter un zéro) ou encore calculer la fonction f(x) = x + 1. Et c'est cette dernière fonction qui va vous être exposée.*
Pour cela, il faudra faire deux actions : positionner la tête de lecture à droite et ensuite effectuer l'addition via le report des bits à 1.

Positionner la tête de lecture :

état	*symbole*	*état*	*mouvement*	*symbole*
début	*0*	*début*	*D*	*0*
début	*1*	*début*	*D*	*1*
début	*B*	*report*	*G*	*B*

Addition (via le report des bits à 1) :

état	*symbole*	*état*	*mouvement*	*symbole*
report	*0*	*stop*	*G*	*1*
report	*1*	*report*	*G*	*0*
report	*B*	*stop*	*G*	*1*

Exécution :

état	*gauche*	*tête*	*droite*
début		*1*	*1011*
début	*1*	*1*	*011*
début	*11*	*0*	*11*
début	*110*	*1*	*1*
début	*1101*	*1*	
début	*11011*		
report	*1101*	*1*	
report	*110*	*1*	*0*
report	*11*	*0*	*00*
stop	*1*	*1*	*100*

Définition 4.42 (T-calculable)**.** *Une fonction f est T-calculable s' il existe une machine de Turing qui, recevant comme donnée n'importe quel nombre entier x fourni tôt ou tard comme résultat $f(x)$ si celui-ci existe.*

Définition 4.43 (T-récursif)**.** *Soit $A \subseteq \mathbb{N}$, A est T-récursif s'il existe une machine de Turing qui, recevant comme donnée n'importe quel nombre naturel x, fournit tôt ou tard comme résultat :* $\begin{cases} 1 & \text{si } x \in A \\ 0 & \text{si } x \notin A \end{cases}$

Définition 4.44 (T-récursivement énumérable)**.** *Soit $A \subseteq \mathbb{N}$, A est T-récursivement énumérable s' il existe une machine de Turing qui, recevant comme donnée n'importe quel nombre naturel x, fourni tôt ou tard comme résultat : 1 si $x \in A$.*
Si $x \notin A$, la machine renvoie un résultat $\neq 1$, s'arrête avec un état $\neq stop$ ou boucle.

4.6.4 Thèse de Church-Turing

1. Toute fonction T-calculable est calculable
2. Toute fonction calculable est T-calculable
3. Tout ensemble T-récursif est récursif
4. Tout ensemble récursif est T-récursif
5. Tout ensemble T-récursivement énumérable est récursivement énumérable
6. Tout ensemble récursivement énumérable est T-récursivement énumérable

Les points 1, 3 et 5 sont des théorèmes. Les autres sont des thèses.

4.6.5 Extension du modèle

On peut modifier le modèle pour changer sa puissance et son efficacité.

Définition 4.45 (Puissance d'une MT)**.** *La puissance d'une MT se mesure en fonction du nombre de fonctions qu'elle peut calculer.*

Définition 4.46 (Efficacité d'une MT)**.** *L'efficacité d'une MT se calcule en fonction du nombre d'instructions à exécuter (on ne tient pas compte de la taille d'un mot mémoire).*

Changer les conventions On peut par exemple permettre de se déplacer de plusieurs cases à la fois ou encore de permettre plusieurs états $stop$.

Influence :

— Même puissance (se déplacer de n cases revient à se déplacer n fois d'une case, on peut donc le programmer avec une MT classique).
— Speedup linéaire (pour aller 20 cases à gauche on doit plus exécuter 20 instructions se déplacer à gauche).

Réduire les symboles Par exemple, ne plus avoir que 0 et 1 comme symboles dans Σ.

Influence :

— Même puissance : chaque symbole dans Σ peut être codé avec des 0 et des 1.
— Même efficacité, car même s' il y a un facteur logarithmique (s'il y a n symboles dans la MT classique, le ruban devra être agrandi de $\log(n)$ pour cette MT modifiée), en calculabilité on le néglige.

Limiter le nombre d'états Cela implique qu'il y a seulement un nombre fini de machines de Turing différentes.

Influence :

— Moins puissant

Autres rubans Ruban unidirectionnel, c'est-à-dire limité d'un coté (à priori à gauche).

Influence :

— Même puissance : on renumérote les cases (voir efficacité) et on ajoute un état de tel sorte que lorsque la tête revient au début, elle repart dans l'autre sens.

— Slowdown linéaire : il faut faire plus de déplacement, en effet, avant les cases étaient numérotés

$$-\infty, ..., -2, -1, 0, 1, 2, ..., +\infty$$

alors que maintenant ce sera

$$0, -1, 1, -2, 2, ..., -\infty, +\infty$$

Ruban multicases La tête lit plusieurs cases en parallèle, ce qui implique que la taille de l'alphabet augmente ($\Sigma \times \Sigma \times ...$).

Influence :

— Même puissance : même idée que précédemment. Si on a une MT à n rubans on peut la transformer en une MT à 1 ruban où la i-ième suite de n cases est associée à la case i dans la MT modifiée. Les états sont construits de telle sorte que la tête lit d'abord les n case de sorte à arriver à un état à la n-ième case où toutes les cases précédentes sont mémorisées. La MT peut alors suivre un état qui prend en compte les n cases précédentes.

— Même efficacité : prendre un alphabet plus grand et une case est équivalent à un plus petit alphabet avec plusieurs cases.

Plusieurs rubans Chaque ruban a sa propre tête. On doit changer la relation de transition, car un état est défini par les positions de toutes les têtes. Le relation doit maintenant prendre l'état (E) et plusieurs symboles ($s_1, ..., s_n$) et retourner un état (E'), plusieurs symboles ($s'_1, ..., s'_n$) à écrire et plusieurs directions différentes, une pour chaque tête ($d_1, ..., d_n$).

$$< s_1, ..., s_n >, E \ \rightarrow \ E', < s'_1, ..., s'_n >, < d_1, ..., d_n >$$

Influence :

— Même puissance : pareil qu'auparavant, sauf qu'il faut ajouter l'information de la position de chaque tête dans l'état. la tête ne doit plus lire une même suite de n cases, mais aller chercher la case dans la bonne suite (exemple : si la tête 3 doit lire la case 5, la tête de la MT classique doit lire la 3e case de la 5e suite de n cases).

— Speedup quadratique

4.6.6 Machine de Turing non déterministe NDT

Tout comme pour les automates non déterministes, on permet plusieurs transitions possibles pour une paire <état, symbole>. La fonction de transition devient une relation de transition, ce qui implique qu'il y a plusieurs exécutions possibles.

Remarque 4.47. *On utilise les NDT uniquement pour décider un ensemble.*

Remarque 4.48. *Cette partie est importante pour la partie concernant la complexité.*

Définition 4.49 (NDT-récursif)**.** *Soit $A \subseteq \mathbb{N}$, A est NDT-récursif s'il existe une ND-machine de Turing telle que lorsqu'elle reçoit comme donnée n'importe quel nombre naturel x :*
Si $x \in A$, alors il existe une exécution fournissant tôt ou tard comme résultat 1.
Si $x \notin A$, alors toutes les exécutions fournissent tôt ou tard comme résultat 0.

Définition 4.50 (NDT-récursivement énumérable)**.** *Soit $A \subseteq \mathbb{N}$, A est NDT-récursivement énumérable s'il existe une ND-machine de Turing telle que lorsqu'elle reçoit comme donnée n'importe quel nombre naturel x :*
Si $x \in A$, alors il existe une exécution fournissant tôt ou tard comme résultat 1.
Si $x \notin A$, toutes les exécutions possibles retournent soit un nombre $\neq 1$,
soit ne se terminent pas, ou encore s'arrêtent avec un état $\neq stop$.

Influence :

— Même puissance, car il existe une machine de Turing qui interprète les NDT.
— Speedup exponentiel, car on "descend" directement au bon endroit dans l'arbre. Mais comme en pratique on doit simuler l'exécution non déterministe par un parcours en largeur de l'arbre d'exécution, ça ne change rien.

4.6.7 Machine de Turing avec Oracle

On ajoute 3 états spéciaux : soit $A \subseteq \mathbb{N}$

— $oracle_{ask}$: demander si l'entier représenté à droite de la tête de lecture appartient à l'ensemble A
— $oracle_{yes}$: l'entier appartient à A
— $oracle_{no}$: l'entier n'appartient pas à A

Puissance : Elle dépend de A. Si A est récursif, ça n'apporte rien et on garde la même puissance, car on peut remplacer l'oracle par un programme qui décide A.

Par contre, si A n'est pas récursif, alors c'est un modèle plus puissant (on pourrait déterminer halt). Mais il n'est pas possible d'exécuter un tel programme.

Remarque 4.51. *Utilité : permet d'établir une hiérarchie parmi les problèmes indécidables. Quels problèmes seraient encore indécidables si K était récursif ?*

4.6.8 Machine de Turing Universelle

Objectif : Construire une machine de Turing qui soit un interpréteur de machines de Turing

Remarque 4.52. *On définit un encodage de 0, 1 qui permet de représenter une MT*

Une telle machine est possible à construire. Il y a plusieurs façons différentes de faire. Une façon intuitive de faire est d'utiliser 3 rubans :

— codage de la MT à interpréter
— donnée
— résultat intermédiaire de l'interpréteur

Mais comme nous l'avons vu plus tôt, utiliser plusieurs rubans ou un seul ruban est identique au niveau de la puissance de calcul.

4.7 Fonctions récursives

Ce modèle de calcul se base sur la définition mathématique de fonction. On va s'intéresser aux fonctions de $\mathbb{N}^k \rightarrow \mathbb{N}$.

Il y a 2 grandes classes de fonctions récursives :

— Fonctions primitives récursives, on se limite aux fonctions totales (équivalent au langage BLOOP)
— Fonction récursives, c'est un modèle complet, on peut calculer toutes les fonctions calculables

Fonctions de bases Ce sont des fonctions qui vont être utilisées pour construire nos fonctions.

Fonctions constantes
$$\begin{array}{|l} a : \mathbb{N}^0 \rightarrow \mathbb{N} \\ a() = a \end{array}$$

Fonctions successeur
$$\begin{array}{|l} s : \mathbb{N} \rightarrow \mathbb{N} \\ s(n) = n + 1 \end{array}$$

Fonctions de projection
$$\begin{array}{|l} p_i^k : \mathbb{N}^k \rightarrow \mathbb{N} \\ p_i^k(x_1, .., x_i, ...x_k) = x_i \end{array}$$

Il existe aussi 2 "règles" importantes :

Composition
$$\begin{array}{|l} h_1, h_2, ..., h_m : \mathbb{N}^k \rightarrow \mathbb{N} \\ g : \mathbb{N}^m \rightarrow \mathbb{N} \\ \overline{x} = x_1, ...x_k \\ f(\overline{x}) = g(h_1(\overline{x}), h_i(\overline{x}), ..., h_m(\overline{x})) \end{array}$$

Récursion primitive
$$\begin{array}{|l} h : \mathbb{N}^{k+2} \rightarrow \mathbb{N} \\ g : \mathbb{N}^k \rightarrow \mathbb{N} \\ \overline{x} = x_1, ..., x_k \\ f(\overline{x}, 0) = g(\overline{x}) \quad \text{(Cas de base)} \\ f(\overline{x}, n+1) = h(\overline{x}, n, f(\overline{x}, n)) \quad \text{(Cas récursif)} \end{array}$$

Remarque 4.53. *Lors de l'utilisation de la récursion primitive, il faut faire attention. Le cas de base ne peut pas faire appel à f et on passe toujours de $n+1$ à n, car il ne peut pas y avoir de récursion infinie.*

4.7.1 Fonctions primitives récursives

Ce modèle ne permet d'utiliser que les fonctions de base et les fonctions obtenues suite à l'application de composition ou de récursion primitive.

Propriété 4.54. *Les fonctions primitives récursives sont calculables.*

Propriété 4.55. *Les fonctions primitives récursives sont des fonctions totales.*

Propriété 4.56. *Les fonctions primitives récursives sont équivalentes aux fonctions calculées par les programmes du langage BLOOP.*

Le langage est dépourvu de boucle.

Propriété 4.57. *Il existe des fonctions totales calculables qui ne sont pas primitives récusrives :*

Par exemple la fonction d'ackermann est calculable, il est possible de coder un programme qui la calcule. Elle est exponentielle. En prenant de grandes valeurs, il est impossible d'expliquer mathématiquement avec des symboles la croissance de cette fonction. [2] Elle ne peut donc pas être exprimée.

Propriété 4.58. *L'interpréteur des fonctions primitives récursives n'est pas une fonction primitive récursive*

Selon le théorème de Hoare-Allison, son interpréteur n'est pas calculable car ce n'est pas une fonction totale calculable.

Propriété 4.59. *Le modèle des fonctions primitives récursives n'est pas un modèle complet de la calculabilité.*

Les fonctions primitives récursives ne sont pas un modèle complet de la calculabilité. En effet, il ne peut pas y avoir de récursion infinie. On ne peut donc calculer avec ce modèle que des fonctions totales calculables.

Exemple 4.60.
Addition *Addition entre deux paramètre, (m + 0) vaut m, Si c'est (m + (n+1)) On utilise la récursivité, on compose avec le successeur. C'est le successeur de l'appel récursif. La complexité est de n.*

$$add(m,0) = m$$
$$add(m,n+1) = s(p_3^3(m,n,add(m,n)))$$

ou plus simplement

$$add(m,0) = m$$
$$add(m,n+1) = s(add(m,n))$$

Multiplication

$$(m \times 0) = 0$$
$$(m \times (n+1)) = m + (m \times n)$$

On multiplie n par m et on rajoute m. La complexité sera ici $(n * m)$, cela revient à faire des $+1$ tout le temps.

$$mult(m,0) = 0$$
$$mult(m,n+1) = add(m, mult(m,n))$$

2. En utilisant la notation des puissances itérées de Knuth, cela est rigoureusement possible.

Soustraction *Revient à prendre un prédécesseur. Complexité de n.*

$$\begin{aligned} pred(0) &= 0 \\ pred(n+1) &= n \\ moins(m,0) &= m \\ moins(m,n+1) &= pred(moins(m,n)) \end{aligned}$$

Comparaison *0 pour 0 sinon 1 pour le positif. m, n booléen signe de m - n*

$$\begin{aligned} signe(0) &= 0 \\ signe(n+1) &= 1 \\ pluspetit(m,n) &= signe(moins(m,n)) \\ egale(m,n) &= moins(1, add(signe(moins(m,n)), signe(moins(n,m)))) \end{aligned}$$

Exemple 4.61. *La fonction d'Ackermann est une fonction calculable* ***non*** *primitive récursive :*

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

Cette fonction a une croissance plus rapide que n'importe quelle fonction primitive récursive :

$$ack(1,m) = m+2, \qquad ack(2,m) = 2m+3, \qquad ack(3,m) = 2^{m+3} - 3$$
$$ack(4,1) \cong 64000, \qquad ack(4,2) \cong 2^{19200}, \qquad ack(4,3) \cong \ldots$$

4.7.2 Fonctions récursives

On va étendre les fonctions primitives récursives en ajoutant une règle :

Minimisation

$h : \mathbb{N}^{k+1} \to \mathbb{N}$
$f : \mathbb{N}^k \to \mathbb{N}$
$\overline{x} = x_1, ..., x_k$
$f(\overline{x}) = \mu_n(h(\overline{x}, n) = 0)$
μ_n est le plus petit n tel que $h(\overline{x}, n) = 0$

Propriété 4.62. *Les fonctions récursives sont un modèle complet de la calculabilité. Toute fonction calculable est une fonction récursive et vice versa.*

4.8 Lambda calcul

Remarque 4.63. *C'est encore un modèle peu intuitif. Je pense que c'est important de refaire l'exercice sur le vrai ou faux en lambda calcul ou encore la représentation des entiers dans le cours. Mais c'est un modèle complet et qui contient la base de la programmation fonctionnelle.*

Remarque 4.64 (Pourquoi le lambda calcul ?)**.** *Pour deux raisons :*

1. *parce que c'est le formalisme de calcul le plus abstrait,*
2. *parce que c'est utilisé dans les logiciels de preuves automatiques.*

Définition 4.65 (Symboles de base)**.** *Soit une variable : $a, b, c, ...y, z, ...$ ou un symbole spécial : $\lambda, (,)$.*

Définition 4.66 (Expression lambda)**.** *Une expression lambda est l'une des 3 choses suivantes :*

— *une variable*

— $\lambda x B$ *si* B *est une expression lambda et que* x *est une variable.*
λx *correspond à la définition d'une variable* ***liée***
$\lambda x B$ *correspond à la définition d'une fonction à un paramètre,* x*.*

— $(F\,A)$ *si* F *et* A *sont des expressions lambda. On dit que* F *est l'opérateur et* A *est l'opérande. Ça représente l'application de* F *à* A*.*

Définition 4.67 (Variable liée)**.** *Une variable liée est une variable qui suit un* λ *ou qui apparait dans* M *et dans* $\lambda x M$*.*

Définition 4.68 (Variable libre)**.** *Une variable libre est une variable qui n'est pas liée.*

4.8.1 Réduction

Objectif : appliquer les fonctions (opérateur) à un opérande, jusqu'à ce qu'il n'y ait plus de fonction à appliquer. On obtient alors une forme réduite au renommage près.

Définition 4.69 (Application de fonction)**.** *Si on a une expression lambda* $(F\,A)$ *où* F *est une fonction* $\lambda x B$*, on remplace toutes les occurrences liées de* x *dans* B *par* A*.*

Définition 4.70 (Au renommage près)**.** *Cela veut dire que les deux expressions suivantes sont équivalentes :*

$$\lambda a(ac) \equiv \lambda x(xy)$$

Remarque 4.71. *Il faut faire attention, car lorsqu'on réduit une expression on ne peut pas introduire de conflit de nom, donc il faut renommer les variables.*

Remarque 4.72. *Il est possible d'avoir des réductions infinies, par exemple :*
$(\lambda x\ (xx)\ \lambda x\ (xx)) \rightarrow (\lambda x\ (xx)\ \lambda x\ (xx))$

Remarque 4.73. *Il est important de voir qu'il y a plusieurs façons de réduire, ça dépend de l'ordre dans lequel on applique les réductions.*

Propriété 4.74. *Une expression lambda est non définie si, peu importe le choix de réduction, on n'arrive pas à une forme réduite.*

Théorème 4.75 (Church-Rosser)**.** *Si 2 séquences de réductions d'une expression lambda conduisent à une forme réduite, alors les expressions obtenues sont équivalentes.*

Propriété 4.76. *Si une forme réduite existe, le choix de réduire l'expression la plus à gauche amène toujours à une forme réduite. (Donc, privilégier la réduction la plus à gauche.)*

Remarque 4.77. *Il existe 2 types de réduction la plus à gauche : la moins imbriquée (semblable au passage par nom en programmation) et la plus imbriquée (semblable au passage par valeur).*

Exemple 4.78.

1	$\lambda a\lambda b((\lambda f\ \lambda cc\ a)\ ((\lambda f\lambda c(fc)\ a)b))$
$\rightarrow$	$\lambda a\lambda b(\lambda cc((\lambda f\lambda c(fc)\ a)b))$
2	$\lambda a\lambda b(\lambda cc((\lambda f\lambda c(fc)\ a)b))$
$\rightarrow$	$\lambda a\lambda b(\lambda cc(\lambda c(ac)\ b))$
3	$\lambda a\lambda b(\lambda cc(\lambda c(ac)\ b))$
$\rightarrow$	$\lambda a\lambda b(\lambda cc(ab))$
4	$\lambda a\lambda b(\lambda cc(ab))$
$\rightarrow$	$\lambda a\lambda b(ab)$

1	*Dans l'expression*	$\lambda f\lambda cc\ a$,	*on remplace toutes les occurrences de*	f	*dans*	λcc	*par*	a.
2	*Dans l'expression*	$\lambda f\lambda c(fc)\ a$,	*on remplace toutes les occurrences de*	f	*dans*	$\lambda c(fc)$	*par*	a.
3	*Dans l'expression*	$\lambda c\ (ac)\ b$,	*on remplace toutes les occurrences de*	c	*dans*	(ac)	*par*	b.
4	*Dans l'expression*	$\lambda c\ c\ (ab)$,	*on remplace toutes les occurrences de*	c	*dans*	c	*par*	(ab).

Définition 4.79 (Représentation des entiers)**.** *Un entier N est défini comme N applications du premier paramètre au second.*

$$\begin{array}{lll} 0 & \Longleftrightarrow & \lambda f \lambda c c \\ 1 & \Longleftrightarrow & \lambda f \lambda c (f c) \\ 2 & \Longleftrightarrow & \lambda f \lambda c (f (f c)) \\ \dots & & \end{array}$$

Ainsi, en utilisant la règle de réduction, si N est la représentation d'un entier.

(Na)	fonction à un paramètre de la forme $\lambda c(a(a(\dots(ac)\dots)))$
((Na)b)	expression de la forme $(a(a(\dots(ab)\dots)))$

Propriété 4.80 (Addition de deux entiers)**.** *Si M et N sont les représentations de deux nombres entiers*

$$\begin{array}{lll} [M+N] & \Longleftrightarrow & \lambda a \lambda b ((Ma)((Na)b)) \\ + & \Longleftrightarrow & \lambda M \lambda N \lambda a \lambda b ((Ma)((Na)n)) \end{array}$$

Exercice 4.81. *Réduire $1 + 2$.*

Chapitre 5

Analyse de la thèse de Church-Turing

Dans ce chapitre on va principalement voir ce qu'est un bon formalisme.

5.1 Fondement de la thèse

Thèse 5.1 (Thèse de Church-Turing)**.** *La forme originale ne contient que 2 parties :*

1. *Toute fonction calculable par une MT est effectivement calculable ;*
2. *Toute fonction effectivement calculable est effectivement calculable par une MT.*

La partie 1 est démontrée (c'est évident puisqu'on peut simuler les machines de Turing).

La partie 2 est supposée vraie : on ne peut pas la démontrer formellement car "effectivement calculable" n'est pas bien défini. On la suppose vraie, car on a des évidences heuristiques (il y a eu beaucoup d'essais pour trouver une fonction qui ne respectait pas la thèse) et qu'on a montré l'équivalence entre tous les formalismes créés à ce jour (on a montré que toute machine constructible par la mécanique de Newton ne calcule que des fonctions calculables).

5.2 Formalismes de la calculabilité

On va se poser la question de ce qui fait un bon formalisme de la calculabilité. Il faut un formalisme qui vérifie les fondements de la thèse. Plus précisément, on va étudier des caractéristiques (des propriétés) nécessaires et suffisantes pour avoir un bon modèle de la calculabilité.

Dans ce chapitre, on va considérer un formalisme de la calculabilité D.

Caractéristiques

SD Soundness [1] des descriptions

CD Complétude des descriptions

SA Soundness algorithmique

CA Complétude algorithmique

U Description universelle

S Propriété S-m-n affaiblie

1. Cohérence

Remarque 5.2. ***D*** *correspond à "description", c'est-à-dire une description de fonction.*
A *correspond à "algorithme", c'est-à-dire à une description exécutable.*

Remarque 5.3. ***Soundness*** *signifie que, si on a une description dans notre modèle D alors celle-ci est cohérente, correspond bien à une fonction calculable.*
Complétude *signifie que notre modèle est complet, qu'il n'existe pas de fonction calculable qui ne le soit pas dans notre modèle.*

Caractéristiques plus en détails

SD Toute fonction D-calculable est calculable (première partie de la thèse de Turing).

CD Toute fonction calculable est D-calculable (deuxième partie de la thèse de Turing).

SA L'interpréteur de D est calculable (on peut exécuter une description de programme de D).

CA Il existe un compilateur qui étant donné un programme p dans un formalisme respectant SA produit une description de programme $d \in D$ tel que p et d calculent la même fonction (cette propriété permet d'assurer l'équivalence entre les formalismes).

U L'interpréteur de D est D-calculable (sinon on sait par Hoare Allison que ce n'est pas un formalisme complet).

S Pour tout programme $d \in$ D à deux arguments, il existe un transformateur de programme S calculable, qui recevant comme entrée le programme d et une valeur x fournit comme résultat un programme $d' = S(d, x)$ tel que $d'(y)$ calcule la même fonction que $d(x, y)$ pour cette valeur de x :

$$\forall d \in D\ \exists S \in D \colon \varphi_d(x, y) = \varphi_{S(d,x)}(y).$$

Un bon formalisme de la calculabilité possède toutes ces propriétés. En pratique, il suffit d'en montrer certaines, car certaines propriétés en entraînent d'autres.

Propriété 5.4. *S et U ⇒ S-m-n.*

Propriété 5.5. *SA ⇒ SD.*

Démonstration. Si l'interpréteur est calculable, en particulier l'interpréteur restreint à un programme est calculable, donc toute fonction D-calculable est calculable. □

Propriété 5.6. *CA ⇒ CD.*

Démonstration. Dans la propriété CA, prenons Java comme formalisme respectant SA (on sait qu'être Java-calculable est équivalent à être calculable). Par CA, tout programme de Java est équivalent à un programme dans D (ils calculent la même fonction), donc toute fonction Java-calculable (c'est-à-dire calculable) est D-calculable. □

Propriété 5.7. *SD et U ⇒ SA.*

Démonstration. Par U, l'interpréteur est D-calculable. Par SD, il est calculable, donc on a SA. □

Propriété 5.8. *CD et S ⇒ CA.*

Démonstration. Considérons n'importe quel autre formalisme P qui a la propriété SA (ce qui signifie que son interpréteur $\mathrm{interpret}_P(n, x)$ est calculable). Donc par la propriété CD, il existe un programme $d \in D$ tel que $\varphi_d(n, x) = \varphi_{\mathrm{interpret}_P}(n, x)$. Or par la propriété S, il existe T un transformateur de programme tel que : $\varphi_d(n, x) = \varphi_{T(n)}(x)$. On a donc :

$$\varphi_n(x) = \varphi_{\mathrm{interpret}_P}(n, x) = \varphi_d(n, x) = \varphi_{T(n)}(x).$$

On peut donc voir T comme un programme qui compile/transforme une description $p \in P$ en une description $d \in D$. Ce qui implique qu'on a bien la propriété CA. □

Un bon formalisme doit donc posséder soit SA et CA ou soit SD, CD, U et S ou soit SA, CD et S ou encore CA, SD et U, car

— SA et CA $\iff$ SD, CD, U et S
— SA, CD et S $\iff$ SD, CA et U

BLOOP et les fonctions primitives récursives sont des "mauvais" formalismes car ils ont seulement les propriétés SD, SA et S.

5.3 Techniques de preuve

Pour prouver qu'un problème est non calculable, on peut utiliser :

— Le théorème de Rice.
— La démonstration directe de la non-calculabilité par diagonalisation ou par preuve par l'absurde. **A priori le plus dur!** C'est plus pratique de réutiliser les problèmes pour lesquels on a déjà montré la non-calculabilité.
— La méthode de réduction.

5.4 Aspects non couverts par la calculabilité

La calculabilité se limite au calcul de fonctions. Or certains problèmes de la vie de tous les jours ne correspondent pas à une fonction.

Exemple 5.9. *Système d'exploitation.*

Exemple 5.10. *Système de réservation aérienne.*

Exemple 5.11. *Certains problèmes utilisent des caractéristiques de l'environnement comme des données en provenance de sonde.*

5.5 Au-delà de la calculabilité

En définissant la calculabilité, on met en évidence des questions qui vont au-delà de son champs d'application :

— Est-il possible d'imaginer un modèle plus puissant que les modèles qu'on a aujourd'hui ?
— Voir même, est-ce que nos modèles de calculabilité sont limités par l'intelligence humaine ?
— Inversement, est-ce que les humains sont "plus puissants" que les machines dans le sens où ils pourraient calculer des fonctions non calculables ?
— Plus précisément dans le cas des machines de Turing : peut-on simuler tout processus mental par une machine de Turing ?

5.5.1 Thèses CT

Les thèses CT sont des approches à la question de l'équivalence entre les fonctions calculables par une machine et les fonctions calculables par un être humain.

Définition 5.12 (H-calculable)**.** *Une fonction est H-calculable si un être humain est capable de calculer cette fonction.*

Les deux thèses suivantes sont deux extrêmes :

Thèse 5.13 (version "réductionniste")**.** *Si une fonction est H-calculable, alors elle est T-calculable car :*

1. *le comportement des élément constitutifs d'un être vivant peut être simulé par une machine de Turing ;*
2. *tous les processus cérébraux dérivent d'un substrat calculable.*

Thèse 5.14 (version "anti-réductionniste")**.** *Certains types d'opérations effectuées par le cerveau, mais pas la majorité d'entre elles (et certainement pas les plus intéressantes !), peuvent être exécutées de façon approximative par les ordinateurs.*
Autrement dit, certains aspects seront toujours hors de protée des ordinateur :

$$\textit{H-calculable} \neq \textit{T-calculable}$$

La thèse suivante cherche à limiter le problème aux fonctions "difficiles à définir" :

Thèse 5.15 (version "procédés publics")**.** *Si une fonction est H-calculable et que l'être humain calculant cette fonction est capable de décrire (à l'aide du langage) à un autre être humain sa méthode de calcul de telle sorte que cet autre être humain soit capable de calculer cette fonction, alors la fonction est T-calculable.*

Remarque 5.16. *Cette thèse n'exclut pas l'existence de fonctions H-calculables, mais non T-calculables ...*

Aucune réponse scientifique à cette question ne peut être donnée, seules différentes positions sont possibles :

— **"Croyant"** : H-calculable $=$ T-calculable (exemple : réductionniste).
— **"Athée"** : H-calculable $\neq$ T-calculable (exemple : anti-réductionniste).
— **"Agnostique"** : pas intéressé par la question.
— **"Iconoclaste"** : la question est mal posée, elle n'a pas de sens.

Lien avec l'IA Deux approches possibles pour l'intelligence artificielle :

1. **IA forte** : simulation du cerveau humain à l'aide d'un ordinateur. En copiant le fonctionnement, on espère ainsi obtenir des résultats similaires.
2. **IA faible** : programmation de méthodes et techniques de raisonnement exploitant les caractéristiques propres des ordinateurs.

La première approche (IA forte) s'apparente à la version "réductionniste" de la thèse de Church-Turing.

Approche par la puissance de calcul On peut tenter de déterminer "l'intelligence" en observant simplement la puissance de calcul :

— Puissance du cerveau humain estimée : entre 10^{13} et 10^{19} instructions par seconde.
— Ordinateurs les plus puissants :
  — IBM ASCI White, Loas Alamos, USA : 13.9 TFLOPS ;
  — Earth Simulator, Japan : $35,8$ TFLOPS (5000 processors).

  Puisque 1 FLO est environ 2 à 10 instructions, cela fait une puissance d'environ 10^{14} instructions par seconde.

Conclusion : les ordinateurs les plus puissants approchent la puissance de calcul du cerveau humain.

5.5.2 Les ordinateurs peuvent-ils penser ?

Le test de Turing (1950) On compare deux tests similaires.

Test de base : 3 personnes, un homme (**A**), une femme (**B**) et un interrogateur communiquent exclusivement via machines à écrire. L'interrogateur posent des questions pour deviner qui est l'homme et qui est la femme. L'homme (**A**) doit faire perdre l'interrogateur, la femme (**B**) le faire gagner.

Test de Turing : une machine (**A**), un être humain (**B**) et un interrogateur communiquent exclusivement via machines à écrire. L'interrogateur posent des questions pour deviner qui est la machine et qui est l'être humain. La machine (**A**) doit faire perdre l'interrogateur, l'être humain (**B**) le faire gagner.

Si l'interrogateur se trompe autant de fois dans le premier test que dans le second, on en conclut que les machines peuvent penser.

Raisonnement de Turing Turing propose le raisonnement suivant pour penser la question "les machines pensent-elles ?" :

- **Hypothèse :** les machines pensent ;
- si on ne peut réfuter l'hypothèse alors celle-ci est vraie ;
- envisager toutes les objections (théologique, émotions, mathématiques...) et toutes les réfuter.

Chapitre 6

Complexité

Lors de l'étude de la complexité, on ne va considérer que la borne supérieure (notation big O). De plus, on ne va considérer que les fonctions totales et donc la décision d'ensembles récursifs. En effet, si la fonction n'est pas totale, l'algorithme peut boucler. Donc on ne sait pas étudier l'efficacité.

Définition 6.1 (Complexité d'un problème)**.** *Complexité de l'algorithme le* ***plus efficace*** *résolvant ce problème.*

Définition 6.2 (Problème pratiquement faisable)**.** *S'il existe un algorithme de complexité polynomiale qui résout ce problème, alors, celui-ci est pratiquement faisable.*

Définition 6.3 (Problème intrinsèquement complexe)**.** *S'il n'existe pas d'algorithme de complexité polynomiale qui résout ce problème, alors, celui-ci est intrinsèquement complexe.*

Remarque 6.4. *Quelle est la différence entre intrinsèquement complexe et pratiquement infaisable ?*

6.1 Influence du modèle de calcul

Si un algorithme est de complexité polynomiale dans un modèle complet alors il sera polynomial dans un autre modèle de calcul. Il y aura juste un facteur polynomial entre les deux, car il existe un compilateur du premier modèle vers le second qui a une complexité polynomiale.

6.2 Influence de la représentation des données

Le choix de représentation de données induit une variation polynomiale du temps d'exécution et de l'espace.

Remarque 6.5. *Certains problèmes sont intrinsèquement complexes juste pour certaines données (simplexe). Celles-ci sont souvent des cas particuliers et peu rencontrées en pratique.*

Chapitre 7

Classes de complexité

On va se "limiter" au problème de décision.

Remarque 7.1. *On ne se limite pas vraiment, car on peut facilement transformer un problème en un problème de décision.*

7.1 Réduction

Objectif : Déduire un algorithme pour un problème P' à partir d'un algorithme P permet :

- de prouver la calculabilité/non calculabilité
- d'analyser le degré de non-calculabilité
- de déduire la complexité
- d'analyser le degré de complexité

Définition 7.2 (Relation de réductibilité)**.** *$A \leq B$: A est réductible à B. Cette relation induit des classes d'équivalence. On peut comprendre ça comme A est plus "simple" que B.*

Il existe plusieurs méthodes de réduction qui ont des propriétés différentes. Mais dans ce cours on va en étudier que 3 :

- réduction algorithmique
- réduction fonctionnelle
- réduction polynomiale

Définition 7.3 (A-complet)**.** *Soit A une classe de problème, un problème E est A-complet* ***par rapport*** *à une relation de réduction $\leq$ si*

1. $E \in A$
2. $\forall B \in A \ : \ B \leq E$

Remarque 7.4. *Le problème E appartient à la classe de problème A et est A-difficile.*

Définition 7.5 (A-difficile)**.** *Soit A une classe de problème, un problème E est A-difficile* ***par rapport*** *à une relation de réduction $\leq$ si*

1. $\forall B \in A \ : \ B \leq E$

Remarque 7.6. *N'importe quel problème de A peut être réduit au problème E, mais E n'appartient pas nécessairement à A.*

7.1.1 Réduction algorithmique

Ce type de réduction n'apporte aucune information au niveau de la complexité, car on peut répéter autant de fois qu'on veut l'algorithme qui décide B. On peut aussi faire un calcul avec une complexité très grande en plus d'utiliser B.

Définition 7.7 (Réduction algorithmique)**.** *Un ensemble A est algorithmiquement réductible à un ensemble B ($A \leq_a B$) si en supposant B récursif, A est récursif.*

Remarque 7.8. *C'est à dire qu'en supposant qu'on connait un algorithme qui décide B, on peut construire un algorithme qui décide A.*

Propriété 7.9. *Si $A \leq_a B$ et B récursif, alors A récursif (par définition)*

Propriété 7.10. *Si $A \leq_a B$ et A non récursif, alors B non récursif (par définition)*

Propriété 7.11. $A \leq_a \overline{A}$

Propriété 7.12. $A \leq_a B \iff \overline{A} \leq_a \overline{B}$

Propriété 7.13. *Si A est récursif alors peu importe le B, $A \leq B$*

Propriété 7.14. *Si $A \leq_a B$ et B récursivement énumérable alors A n'est* ***pas nécessairement récursivement énumérable***

7.1.2 Réduction fonctionnelle

Ce type de réduction n'apporte aucune information au niveau de la complexité, car tout dépend de la complexité de la fonction f.

Définition 7.15 (Réduction fonctionnelle)**.** *Un ensemble A est fonctionnellement réductible à un ensemble B ($A \leq_f B$) s'il existe une fonction* ***totale calculable*** *f telle que*

$$a \in A \iff f(a) \in B$$

Remarque 7.16. *Donc pour décider si $a \in A$ il suffit de calculer $f(a)$ et décider si $f(a) \in B$. Pour trouver une réduction fonctionnelle, il faut trouver une fonction qui transforme un problème de A en un problème de B.*

Propriété 7.17. *Si $A \leq_f B$ et B récursif, alors A récursif (par définition)*

Propriété 7.18. *Si $A \leq_f B$ et A non récursif, alors B non récursif (par définition)*

Propriété 7.19. $A \leq_f B \iff \overline{A} \leq_f \overline{B}$

Propriété 7.20. *Si A est récursif alors peu importe le B, $A \leq B$*

Propriété 7.21. *Si $A \leq_f B$ et B récursivement énumérable alors A est* ***nécessairement récursivement énumérable***

Propriété 7.22. $A \leq_f B \Rightarrow A \leq_a B$ *(Attention ce n'est pas toujours vrai dans l'autre sens)*

7.1.3 Différence entre $\leq_a$ et $\leq_f$

La principale différence, c'est que $A \leq_a B$ est plus du point de vue de la calculabilité. On s'intéresse au fait que ça soit possible, on peut utiliser autant de fois que l'on veut le fait que B soit récursif.
Alors que $A \leq_f B$ est plus du point de vue de la complexité. On est obligé d'utiliser un certain schéma d'algorithme :

```
input a
// some work
a2 := f(a)
// some work
if a2 in B then 1
else 0
```

On est donc limité à utiliser qu'une fois le test $f(a) \in B$ **en dernier lieu**.

7.2 Modèles de calcul

D'habitude, on utilise les machines de Turing pour avoir une définition précise de la complexité. Mais c'est peu intuitif et on s'intéresse aux frontières. Or la différence de complexité entre différents modèles est un facteur polynomial (c'est une thèse) ce qui n'a pas d'influence sur les frontières.

7.3 Classes de complexité

Classes basées sur le modèle déterministe

Définition 7.23 (DTIME(f))**.** *Famille des ensembles récursifs pouvant être décidés par un programme Java de complexité temporelle $\mathcal{O}(f)$*

Définition 7.24 (DSPACE(f))**.** *Famille des ensembles récursifs pouvant être décidés par un programme Java de complexité spatiale $\mathcal{O}(f)$*

Classes basées sur le modèle non déterministe

Définition 7.25 (NTIME(f))**.** *Famille des ensembles récursifs pouvant être décidés par un programme non déterministe Java de complexité temporelle $\mathcal{O}(f)$*

Remarque 7.26. *On considère juste la complexité de la branche d'exécution la plus longue. Toutes les branches sont donc finies.*

Définition 7.27 (NSPACE(f))**.** *Famille des ensembles récursifs pouvant être décidés par un programme non déterministe Java de complexité spatiale $\mathcal{O}(f)$*

Définition 7.28 (Classe P)**.**

$$P = \bigcup_{i \geq 0} DTIME(n^i)$$

Famille des ensembles récursifs pouvant être décidés par un programme Java de complexité temporelle polynomiale.

Remarque 7.29. *Les classes ne dépendent pas du modèle de calcul*

Définition 7.30 (Classe NP)**.**

$$NP = \bigcup_{i \geq 0} NTIME(n^i)$$

Famille des ensembles récursifs pouvant être décidés par un programme Java non déterministe de complexité temporelle polynomiale.

Remarque 7.31. *Si on savait faire du non-déterminisme, on aurait une complexité polynomiale, mais pour le moment on ne peut que le simuler donc on a une complexité exponentielle.*

7.4 Relations entre les classes de complexité

Déterministe vs non-déterministe

Propriété 7.32. $A \in NTIME(f) \Rightarrow A \in DTIME(c^f)$
f est la profondeur maximale de l'arbre. Si on simule le ND, alors on doit faire un bfs dans un arbre de profondeur f. c est le facteur de branchement.

Propriété 7.33. $A \in NSPACE(f) \Rightarrow A \in DSPACE(f^2)$
C'est une borne sur le nombre de nœuds d'un graphe de profondeur f. Théorème de Savitch.

Time vs Space

Propriété 7.34. $A \in DTIME(f) \Rightarrow A \in DSPACE(f)$
Le programme ne peut utiliser qu'au maximum un emplacement mémoire par instruction. Donc l'espace utilisé est limité par le nombre d'instructions.

Propriété 7.35. $A \in NTIME(f) \Rightarrow A \in NSPACE(f)$
C'est la même chose que pour le cas déterministe (propriété précédente).

Propriété 7.36. $A \in DSPACE(f) \Rightarrow A \in DTIME(c^f)$

Démonstration. On sait que le programme se termine pour tout car l'ensemble est récursif. Si la mémoire utilisée est en $O(f)$, le nombre de configurations possibles de cette mémoire (stack, heap, program counter, ...) est exponentiel $O(c^f)$. Si la complexité temporelle dépasse cette complexité exponentielle, alors l'exécution du programme va repasser sur un état de mémoire déjà rencontré, et va y repasser une infinité de fois. Le programme va alors ne pas se terminer, ce qui est contraire à l'hypothèse que l'ensemble est récursif. □

Propriété 7.37. $A \in NSPACE(f) \Rightarrow A \in NTIME(c^f)$
C'est la même chose que pour le cas déterministe (propriété précédente).

Hiérarchie de complexité

On peut prouver qu'il existe pire qu'une complexité exponentielle.

7.5 NP-complétude

La question fondamentale de la complexité est la suivante : **s'il existe un algorithme non déterministe polynomial, existe-t-il un algorithme déterministe polynomial résolvant ce même problème ?** C'est-à-dire, est-ce que $P = NP$ **?** On sait que $P \subseteq NP$ mais on n'a pas encore montré si oui ou non $NP \subseteq P$.

Pour démontrer ça, on essaye de montrer qu'un élément de NP, le plus difficile, est dans P. On choisit un élément qui soit NP-complet par rapport à une relation de réduction. Ainsi, si on y arrive, ça implique que tous les autres éléments de NP sont dans P aussi.

La question est maintenant de choisir la relation de réduction. Les 2 relations de réduction définies précédemment ne suffisent pas. Ces réductions ne permettent pas d'affirmer quelque chose sur la complexité. On introduit donc une nouvelle réduction, la réduction polynomiale.

Définition 7.38 (Réduction polynomiale)**.** *Un ensemble A est polynomialement réductible à un ensemble B, $A \leq_p B$ s'il existe une fonction* ***totale calculable*** *f de* ***complexité temporelle polynomiale*** *telle que*

$$a \in A \Leftrightarrow f(a) \in B$$

Remarque 7.39. *On ajoute à la réduction fonctionnelle une contrainte de complexité sur la fonction f. Cette réduction nous permet donc de tirer des conclusions sur la complexité de A connaissant la complexité de B.*

Propriété 7.40.

$$A \leq_p B \text{ et } B \in P \Rightarrow A \in P$$

Ce qui est logique étant donné qu'on a la complexité de f qui est polynomiale + la complexité de la décision de B qui est aussi polynomiale.

Propriété 7.41.

$$A \leq_p B \text{ et } B \in NP \Rightarrow A \in NP$$

Ce qui est logique étant donné qu'on à la complexité de f qui est polynomiale + la complexité de la décision de B qui est non déterministe polynomiale.

Définition 7.42 (NP-complétude)**.** *Un problème E est NP-complet (par rapport à $\leq_p$) si :*

1. $E \in NP$
2. $\forall B \in NP : B \leq_p E$

Propriété 7.43. *Soit E un ensemble NP-complet,*

$$E \leq_p B \textbf{ et } B \in NP \Rightarrow B \text{ est } NP\text{-complet}$$

Ce qui est logique puisque ça veut dire que B est NP-difficile et dans NP.

En supposant $P \neq NP$, on peut représenter les différents classes de problèmes comme sur la figure 7.1

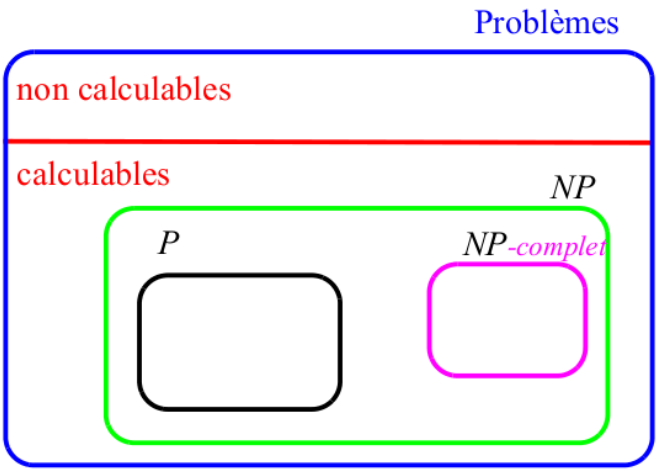


FIGURE 7.1 – Classes de problèmes (en supposant $P \neq NP$)

Nous allons désormais essayer de trouver des problèmes NP-complets et de trouver des propriétés intéressantes sur P.

7.5.1 Problème de décision

On va définir différemment la classe NP. On va considérer des problèmes de décision. Pour un ensemble A cela consiste à dire si oui ou non une donnée x appartient à A. On peut voir cela comme un prédicat. Par exemple $SAT(x)$: la formule x est-elle satisfaisable ?

Redéfinition de P et NP en problème de décision.

Définition 7.44 (Classe P)**.** *La classe P est la classe des problèmes de décision pouvant être décidés par un algorithme polynomial.*

Définition 7.45 (Classe NP)**.** *La classe NP est la classe des problèmes de décision $A(x)$ pouvant s'exprimer sous la forme $\exists y\ B(x, y)$ tel que :*

- *$B(x, y) \in P$ (Il est donc facile de vérifier une solution)*
- *le domaine de y est fini (taille polynomiale en x) et peut être généré, de manière non déterministe, en un temps polynomial*

Remarque 7.46. *On peut se représenter ça comme si le non-déterminisme permettait de générer tous les y "en même temps" ou que le non-déterminisme choisissait le bon y. Il est rapide de tester une solution, mais pas d'en trouver une.*

Définition 7.47 (Calcul d'un problème NP)**.** *Pour décider $A(x)$*

1. *calculer y (de manière non déterministe)*
2. *déterminer $B(x, y)$*

7.6 Théorème de Cook : SAT est NP-complet

Pour pouvoir trouver des problèmes NP-complet en les réduisant par rapport à un problème NP-complet, il faut trouver un premier problème NP-complet.

On va montrer que SAT est NP-complet en 2 parties :

1. $SAT \in NP$
2. $\forall B \in NP\ :\ B \leq_p SAT$

7.6.1 Le problème SAT

Le problème $SAT(x)$ est de décider si la formule propositionnelle x est satisfaisable ou non. C'est-à-dire : est-ce que $x \in SAT$?

On définit l'ensemble SAT comme l'ensemble des formules propositionnelles satisfaisables.

Définition 7.48 (Formule propositionnelle satisfaisable)**.** *Soit n variables A_i booléennes, et une série de connecteurs logiques $\neg$ (non), $\wedge$ (et), $\vee$ (ou), $\Rightarrow$, $\Leftrightarrow$, (,). Une formule propositionnelle $x(A_1, ..., A_n)$ est satisfaisable s'il existe des valeurs logiques (true ou false) pour les variables $A_1, ..., A_n$ tel que $x(A_1, ..., A_n)$ soit vraie.*

La longueur d'une formule x est $O(n \log n)$, où n est le nombre d'occurrences des variables. Ça se justifie par le fait qu'en utilisant un codage d'Huffman, le code pour une variable prendra $\log n$ (c'est important pour définir la complexité du problème).

Exemple 7.49. *Soit les trois variables A, B et C*

$$x(A, B, C) = (\neg A \vee B) \wedge (B \vee C) \wedge (A \vee \neg C)$$

$(\neg A \vee B)$, $(B \vee C)$ et $(A \vee \neg C)$ sont les trois clauses de notre formules. Elles ont chacune deux litéraux. La formule est ici satisfaite si $A = false$, $B = true$ et $C = false$. D'autres solutions sont possibles.

7.6.2 $SAT \in NP$

Il existe un programme ND polynomiale capable de décider si $x \in SAT$. On pose que m est le nombre de variables de x et n est le nombre d'occurrences de variables ($m \leq x$). Étapes de l'algorithme avec leur complexité :

— Générer une séquence de m valeurs logiques de façon non déterministe : $\mathcal{O}(m)$

— Substituer les occurrences des variables par leur valeur : $\mathcal{O}(n \log n)$

— Évaluer l'expression : Complexité polynomiale par une technique de réduction.

7.6.3 $\forall B \in NP \ : \ B \leq_p SAT$

Comme $B \in NP$, on a une NDMT qui décide B en un temps polynomial $p(n)$. On va montrer qu'on sait transformer en un temps polynomial la NDMT par rapport à n en une formule propositionnelle et que cette formule à une longueur qui dépend de $p(n)$ ($\mathcal{O}(p(n))$ symboles). Ce qui prouve que $B \leq_p SAT$.

Idée de la transformation Il me semble qu'on ne doit pas la connaitre pour l'examen. Mais l'idée est de représenter le ruban comme un tableau de variables booléennes (chaque ligne représente le ruban à un instant), même chose pour les états, le curseur, l'alphabet,...

7.7 Quelques problèmes NP-complets

— Problème du circuit hamiltonien HC (trouver un chemin qui passe une seule fois par tous les sommets)

— Problème du voyageur de commerce TS (trouver un chemin qui relie tous les sommets et de longueur $\leq B$)

— Chemin le plus long entre 2 sommets dans un graphe

— $3SAT$ (forme conjonctive avec 3 variables par clause)

— Programmation entière (simplexe)

— ...

Chapitre 8

Solutions exercices

8.1 Introduction

8.2 Concepts

Exercice 2.11.
Posons E_k l'ensemble des nombres algébriques qui satisfont le polynôme :
$a_0 + a_1x + a_2x^2 + ... + a_nx^n$. avec n<k et max($|c_j|$) < k.
Il existe au plus k^k polynômes de cette forme, et chacun a au plus k racines.
Donc E_k est un ensemble fini ayant au plus k^{k+1} éléments. Ainsi nous listons les nombres algébriques E_1, E_2, E_3, E_4 et retirons les répétitions et obtenons une nouvelle ensemble qui est enumerable.

Exercice 2.26.

Exercice 2.27.
Soit A l'ensemble des nombres algébriques et T l'ensemble des nombres transcendantaux.
Notons que l'ensemble $R = A \cup T$ et A est énumérable. Si T étais énumérables, alors R serait l'union de deux ensembles énumérables. Puisque R est non-énumérable, par conséquent, T est non-énumérable cas s'il etais R serait énumérable.

8.3 Resultats Fondamentaux

8.4 Modèles de calculabilité

Exercice 4.81.
Représentons d'abord 1 et 2 en expressions lambda.

$$\begin{aligned} 1 &\iff \lambda f \lambda c(fc) \\ 2 &\iff \lambda f \lambda c(f(fc)) \end{aligned}$$

Par la règle de l'addition (4.80), nous trouvons

$$\begin{aligned} [1+2] &\iff \lambda a \lambda b(((\lambda f \lambda c(fc))a)(((\lambda f \lambda c(f(fc)))a)b)) \\ &\iff \lambda a \lambda b((\lambda c(ac))((\lambda c(a(a(c))))b)) \\ &\iff \lambda a \lambda b((\lambda c(ac))((a(a(b))))) \\ &\iff \lambda a \lambda b(a(a(a(c)))) \end{aligned}$$

Ce qui est bien la représentation lambda de 3.

8.5 Analyse de la thèse de Church-Turing

8.6 Complexité

8.7 Classes de complexité