Table des matières

Introduction

L'objectif dans ce mémoire Master est l'étude de quelques notions sur la calculabilité que les anglo-saxons désignent par "Theory of computation". Cette théorie s'intéresse géneralement aux trois questions fondamentales suivantes:

1. Quels problèmes peut-on (pratiquement) résoudre à l'aide d'un ordinateur ? telle est la question centrale que pose la théorie de calculabilité.

2. Existe-t-il un meilleur algorithme pour un problème démontré soluble? On peut comparer les algorithmes entre eux, on a ce qu'on appelle une hiérarchie de complexité.

3. Existe-t-il un algorithme pour résoudre tel ou tel problème ? c'est la question que pose la théorie de la décidabilité.

Dans ce travail on s'intéresse à l'étude d'un modèle de calcul qui est les fonctions récursives dont l'origine débute depuis les années 1930 par l'étude de la notion de processus effectif de calcul. Différents modèles équivalents pour cette notion ont vue le jour: à l'aide des fonctions récursives par Gödel et Kleene, le λ calcul par Church (1936), les systèmes de post (1936-37) et enfin les Machines de Turing par Turing (1936-37). Dans ce mémoire, on s'intéresse à l'étude des fonctions récursives qui sont obtenu à partir des fonctions de base et les constructeurs : composition, récursion et la minimisation.

Notre travail est organisé en quatre chapitres:

1. Dans le premier chapitre nous donnons quelques notions sur la calculabilité.

2. Dans le deuxième chapitre nous étudions les fonctions et les prédicats primitifs récursifs.

3. Le troisième chapitre est consacré à l'étude d'une fonction non primitive récursive, qui est la fonction d'Ackremann.

4. Dans le quatrième chapitre nous étudions un autre modèle de calcul qui est la machine de Turing.

Chapitre 1

Notions sur la calculabilité

Dans ce chapitre, nous donnons les principaux résultats de la théorie de la calculabilité. La théorie de la calculabilité permet de discuter l'existence, ou la non-existence d'algorithmes pour résoudre un problème donné.

1.1 Définitions formelles de la notion d'algorithme et modèles de calcul

Notre but est de donner une définition formelle de la classe des fonctions calculables. Cette définition doit être à la fois intuitive et correspondre à la pratique de programmation.

Une définition simple est la suivante: une fonction f est calculable s'il existe un algorithme pour la calculer.

L'approche, que nous allons utiliser dans ce chapitre, est dite fonctionnelle. Nous allons restreindre notre attention aux algorithmes pour calculer les fonctions de la forme $f : \mathbb{N}^k \rightarrow \mathbb{N}$, et nous voulons aboutir à une définition (formelle) des fonctions calculables $f : \mathbb{N}^k \rightarrow \mathbb{N}$.

Pour voir le lien entre la décidabilité et la calculabilité, considérons un problème de décision (U, B) où $U = \mathbb{N}^k$. Le problème est donc le suivant: pour $x \in U$ décider si $x \in B$. Afin de décider si une entrée $x \in B$, nous allons utiliser la fonction caractéristique de l'ensemble B $\chi_B(x) : U \rightarrow \mathbb{N}$, qui est définie comme suit.

Définition 1.1.1 ***Fonction caractéristique***

$$\chi_B(x) = \begin{cases} 1 & si \quad x \in B \\ 0 & sinon \end{cases}$$

Le problème (U, B) est décidable si et seulement si χ_B est calculable.

Une fois que nous aurons une notion précise de fonction calculable, la définition précédente deviendra aussi rigoureuse.

– Le problème P est décidable , Il existe un algorithme pour P (c'est-à-dire une procédure qui s'arrête et répond "OUI" si l'entrée $x \in B$ et répond "NON" si l'entrée $x \notin B$).

– Le problème P est indécidable , Il n'existe pas d'algorithme pour résoudre P.

– Le problème P est semi-décidable , Il existe un semi-algorithme pour résoudre P (c'est-à-dire une procédure telle que si l'entrée $x \in B$ elle répond "OUI" ; si l'entrée $x \notin B$ dit "NON" ou ne donne pas de réponse).

Il est clair qu'un problème décidable est aussi semi-décidable.

- Pour montrer qu'un problème est décidable, il suffit de trouver un algorithme (un seul suffit et ceci peut se faire sans utiliser la théorie de la calculabilité).

- Par contre, pour montrer qu'un problème est indécidable, il faut considérer tous les algorithmes possibles et montrer qu'aucun d'eux ne résout pas le problème, ce qui est plus difficile et impossible sans théorie et sans notion rigoureuse d'algorithme.

- Nous représentons une brève comparaison entre ces deux disciplines sous la forme d'un tableau

	Calculabilité	Complexité
Question	Existe-t-il un algorithme ?	Existe-t-il un algorithme rapide ?
Hypothèse	Ressources non-bornées	Ressources bornées
Technique 1	Diagonalisation	Diagonalisation
Technique 2	Réduction calculable	Réduction polynômiale

1.1.1 Un langage de programmation

Le développement de la théorie du calculabilité sera basée sur un langage de programmation spécifique $\mathcal{L}$. Nous allons utiliser certaines lettres comme des variables dont les valeurs sont

des nombres, en particulier, les lettres $X_1, X_2, X_3...$ seront appelés *variales d'entrée* de $\mathcal{L}$, la lettre Y sera appelé *variable de sortie* de $\mathcal{L}$, et les lettres $Z_1, Z_2, Z_3...$ seront appelés *variables locales* de $\mathcal{L}$.

En $\mathcal{L}$, nous serons en mesure d'écrire l'instruction de toutes sortes, un programme de $\mathcal{L}$ consistera ensuite une suite (une séquence finie) d'instructions. Par exemple, pour chaque variable V, il y aura une instruction:

$$V \leftarrow V + 1$$

$V \leftarrow V + 1$ C'est-à-dire augmenter par 1 la valeur de la variable V.

$V \leftarrow V - 1$ C'est-à-dire si la valeur de V est égal à 0, le laisser inchangé, sinon diminue de 1 la valeur de V.

$if\ V \neq 0\ goto\ L$ C'est-à-dire si la valeur de V est différent de zéro exécuter l'instruction avec l'étiquette L prochaine, sinon procéder à l'instruction suivante dans la liste.

Ces instructions seront appelés respectivement les instructions de branchement incrémenter, décrémenter et conditionnel.

Nous allons utiliser la convention spéciale que la grandeur de sortie Y et Z_i les variables locales ont initialement la valeur 0. Nous allons parfois indiquer la valeur d'une variable en l'écrivant en italiques minuscules. Ainsi, x_5 est la valeur de X_5.

Instructions peuvent ou peuvent ne pas avoir l'étiquette. Quand une instruction est étiqueté, l'étiquette est écrit à sa gauche entre crochets. Par exemple, $[B] \qquad Z \leftarrow Z - 1$.

1.1.2 Quelques exemples

1. Le programe qui calculer la fonction $f(x_1, x_2) = x_1 + x_2$ est la suivant:

$$\begin{array}{ll} & Y \leftarrow X_1 \\ & Z \leftarrow X_2 \\ [B] & if\ Z \neq 0\ goto\ A \\ & goto\ E \\ [A] & Z \leftarrow Z - 1 \\ & Y \leftarrow Y + 1 \\ & goto\ B \end{array}$$

2. Le programe qui calculer la fonction $f(x_1, x_2) = x_1.x_2$ est la suivant:

$$
\begin{array}{ll}
 & Z_2 \leftarrow X_2 \\
[B] & if\ Z_2 \neq 0\ goto\ A \\
 & goto\ E \\
[A] & Z_2 \leftarrow Z_2 - 1 \\
 & Z_1 \leftarrow X_1 + Y \\
 & Y \leftarrow Z_1 \\
 & goto\ B
\end{array}
$$

Chapitre 2

Fonctions et prédicats primitifs récursifs

2.1 Fonctions primitives récursives

Nous donnons dans ce chapitre la première classe de fonctions calculables. On utilisons les *fonctions de base* O, σ et π, et les *Constructeurs* $Comp$ et Rec (c-à-d les mécanisemes de construction de nouvelle fonctions à partir des fonctions de base) pour obtenir une classe assez large, appelée *fonctions récursives primitives* (RP), contenant presque toutes les fonctions que nous connaissons, mais, néeanmoins insuffisante.

La classe des fonctions récursives primitives (RP) est la classe de fonctions

$f : \mathbb{N}^k \to \mathbb{N}$ définie inductivement en utilisant les trois fonctions de base et les deux constructeurs qui définissent comme suit:

Définition 2.1.2 ***Fonctions de base***

Les focntions primitives récursives de base sont les suivantes.

1. La fonction Zéro $O : \mathbb{N} \to \mathbb{N}$ telle que:

$$O(n) = 0$$

2. La fonction Successeur. Elle est défie par $\sigma : \mathbb{N} \to \mathbb{N}$ telle que:

$$\sigma(n) = n + 1$$

3. La fonction de Projection. Elle est défie par $\pi_i^k : \mathbb{N}^k \to \mathbb{N}$ telle que:

$$\pi_i^k(n_1, n_2, ..., n_k) = n_i.$$

Définition 2.1.3 ***Constructeurs***

1. Composition ($Comp$)

   Soit g une fonction à l arguments et $h_1, ..., h_l$ des fonction à k arguments. Si nous dénotons $n_1, ..., n_k$ par $\overline{n}$, alors la composition de g et de $h_1, ..., h_l$ est la fonction $f : \mathbb{N}^k \longrightarrow \mathbb{N}$ définie par :

$$f(\overline{n}) = g(h_1(\overline{n}), h_2(\overline{n}), ..., h_l(\overline{n})).$$

2. Récursion primitive ($Rec\ pri$)

Soit h une fonction à k arguments et g une fonction à $k+2$ argument. Alors, la fonction f à $k+1$ telle que:

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

est la fonction définie à partir de g et h par récursion primitive.

La classe (RP) est la plus petite classe qui contient les fonctions de base o, σ, π_i^k et est fermée par les constructeurs $Comp$ et $Rec\ pri$.

Exemple 2.1.4 *Nous allons définir la fonction $Plus(x, y) = x + y$. D'abord*

$$\begin{cases} Plus(x, 0) = x \\ Plus(x, n+1) = Plus(x, n) + 1 \end{cases}$$

Il suffit maintenant de reécrire les équations ci-dessus sous forme $Rec(h, g)$. Nous avons

$$\begin{cases} Plus(x, 0) = \pi_1^1(x) = f(x) \\ Plus(x, n+1) = \sigma(\pi_3^3(x, n, Plus(x, n))) = g(x, n, Plus(x, n)) \end{cases}$$

donc, $Plus = Rec(\pi_1^1, Comp(\sigma, \pi_3^3))$.

Exemple 2.1.5 *zero (avec un argument).*

$$\begin{cases} zero(0) = 0 = O(n) = f(n) \\ zero(n+1) = zero(n) = \pi_2^2(n, zero(n)) = g(n, zero(n)) \end{cases}$$

donc, $zero = Rec(O, \pi_2^2)$.

2.1.1 Réaliser une fonction en un langage

Programmer les fonctions de base. En ce qui concerne la fonction *Comp*, si l'on sait programmer $f(x_1, ..., x_k)$ et $g_1(x_1, x_2, ..., x_k), ..., g_n(x_1, x_2, ..., x_k)$, on peut alors programmer $f(x_1, ..., x_k) = h(g_1(x_1, ..., x_k), ..., g_n(x_1, ..., x_k))$.

Pour la fonction *Comp*, un programme pour la calculer est le suivant.

$$\begin{aligned} & Z_1 \leftarrow g_1(X_1, ..., X_k) \\ & Z_2 \leftarrow g_2(X_1, ..., X_k) \\ & \vdots \\ & Z_k \leftarrow g_k(X_1, ..., X_k) \\ & Y \ \leftarrow h(Z_1, ..., Z_k) \end{aligned}$$

Pour la fonction *Recpri* qui est définie par: $f(x_1, x_2, ..., x_k, 0) = h(x_1, x_2, ..., x_k)$ et $f(x_1, x_2, ..., x_k, m+1) = g(x_1, x_2, ..., x_k, m, f(x_1, x_2, ..., x_k, m))$

un programme pour calculer $f(x_1, x_2, ..., x_k, x_{k+1})$ est le suivant.

$$\begin{aligned} & Y \leftarrow h(X_1, ..., X_k) \\ [A] \qquad & if\ X_{k+1} + 0\ goto\ E \\ & Y \leftarrow g(Z,\ Y,\ X_1, ..., X_k) \\ & Z \leftarrow Z + 1 \\ & X_{k+1} \leftarrow X_{k+1} - 1 \\ & goto\ A \end{aligned}$$

2.1.2 Quelques propriétés de fermeture de la classe des fonctions primitives récursives

Décrire les fonctions RP par des termes ne contenant que des fonctions de base et les constructeurs $Comp$ et Rec est une tâche fastidieuse. Nous établirons plusieurs propriétés de fermeture de la classe RP qui nous permettront d'établir la récursivité partielle plus rapidement.

Considérons $f(n) = \sum_{i=0}^{n} i!$. Afin de montrer que cette fonction est primitive récursive, notons d'abord que $i!$ est RP puisqu'on peut l'exprimer comme suit:

$$\begin{cases} 0! = 1 \\ (i+1)! = (i+1).i! \end{cases}$$

Ensuite, f peut être écrite comme suit: $\begin{cases} f(0) = 1 \\ f(n+1) = f(n) + (n+1)! \end{cases}$

la fonction f est donc primitive récursive.

Si $g(x,i)$ est primitive récursive, alors $f(x,n) = \sum_{i=0}^{n} g(x,i)$ et $h(x,n) = \prod_{i=0}^{n} g(x,i)$ sont aussi RP.

Autrement dit, la classe de fonctions RP est fermée par rapport à $\sum$ et $\prod$.

2.1.3 Quelques exemples de fermeture de la classe des fonctions primitives récursives

Exemple 2.1.6 ***L'addition***

La fonction d'addition $plus(n_1, n_2)$ peut être définie par récursion primitive :

$$\begin{cases} plus(n_1, 0) = \pi_1^1(n_1) \\ plus(n_1, n_2 + 1) = \sigma(\pi_3^3(n_1, n_2, plus(n_1, n_2))) \end{cases}$$

Exemple 2.1.7 ***Prédécesseur***

La fonction prédécesseur p est définie par $p(n) = max(0, n-1)$ est primitive récursive. Elle peut être définie de la manière suivante:

$$\begin{cases} p(0) = 0 = O(n) \\ p(n+1) = n \\ p(n+1) = n = \sigma(\pi_2^2(n, p(n))) = g(n, p(n)) \end{cases}$$

donc $p = Rec(\sigma, \pi_2^2)$

Exemple 2.1.8 ***différence***

La fonction sub définie par: $\begin{cases} sub(n,m) = n - m & \text{si} \quad n \geq m \\ 0 & \text{sinon} \end{cases}$

est aussi primitive récursive

elle peut être définie de la manière suivante:

$$\begin{cases} sub(n, 0) = n \\ sub(n, 0) = n = \pi_1^1(n) = f(n) \\ sub(n, m+1) = p(sub(n, m)) \end{cases}$$

Exemple 2.1.9 ***Produit***

La fonction $prod$ définie par: $prod(n, m) = n * m$ est primitive récursive. Elle peut être définie de la manière suivante:

$$\begin{cases} prod(n, 0) = 0 \\ prod(n+1, m) = plus(prod(n, m), m) \end{cases}$$

Exemple 2.1.10 ***Signe***

On définit la fonction sg de la façon suivante :

$$sg(m) = \begin{cases} 0 & \text{si} \quad m = 0 \\ 1 & \text{si} \quad m > 0 \end{cases}$$

2.2 Prédicats primitifs récursifs

Définition 2.2.11

Un prédicat est une fonction dont les valeurs sont prises dans l'ensemble $\{vrai, faux\}$. L'ensemble des prédicats définis sur l'ensemble $\mathbb{N}^k$ est donc

$$\left\{\mathbb{N}^k \rightarrow \{vrai, faux\} \ / \ k \geq 0\right\}.$$

De façon équivalente, un prédicat défini sur $\mathbb{N}^k$ peut être défini par un sous-ensemble de $\mathbb{N}^k$ (l'ensemble des éléments de $\mathbb{N}^k$ pour lequel le pédicat est vrai).

Un prédicat P à k arguments est un sous-ensemble de $\mathbb{N}^k$ (les éléments de $\mathbb{N}^k$ pour lesquels P est vrai).

Un prédicat sur $\mathbb{N}^k$ d'écrit une propriété d'un $k-uplet$ $(x_1, ..., x_k)$. Il peut être défini par une fonction $\mathbb{N}^k \rightarrow \{vrai, faux\}$ ou bien par un sous-ensemble de $\mathbb{N}^k$.

Exemple 2.2.12

Le prédicat $Pair$ sur $\mathbb{N}$. $Pair(n)$ est vrai si son argument est un nombre pair, et faux sinon. Il correspond au sous ensemble des entiers naturels composé de tous les nombres pairs.

Exemple 2.2.13

Le prédicat $PlusGrand$ sur $\mathbb{N}^2$ est défini comme suit :

$$PlusGrand(x, y) = \begin{cases} vrai & \text{si } \ x > y \\ faux & \text{sinon} \end{cases}$$

il correspond à un ensemble $\{(1,0), (2,0), ..., (2,1), (3,1), ...\}$.

Exemple 2.2.14

Considérons le prédicat $zerop$, qui est vrai pour la valeur 0. Sa définition est la suivante:

$$\begin{cases} zerop(0) = 1 \\ zerop(n+1) = 0 \end{cases}$$

Dans le cadre défini dans l'introduction un prédicat P sur $\mathbb{N}^k$ correspond au problème (U, B) avec $U = \mathbb{N}^k$ et $B = \{x/P(x) \text{ est vrai}\}$.

2.2.1 Fonction caractéristique d'un prédicat

Définition 2.2.15

La fonction caractéristique d'un prédicat $P \subseteq \mathbb{N}^k$ est la fonction $\chi_P : \mathbb{N}^k \to \{0,1\}$ telle que

$$\chi_P(\overline{n}) = \begin{cases} 1 & \text{si} \quad \overline{n} \in P \\ 0 & \text{si} \quad \overline{n} \notin P \end{cases}$$

La notion de pridicat primitif récursif est alors directement définie à partir de la notion de fonction caractéristique.

Exemple 2.2.16

La fonction caractéristique du prédicat $Pair$ définie sur l'ensembles $\mathbb{N}$ est donc

$$\begin{cases} \chi_{Pair}(2n) = 1 \\ \chi_{Pair}(2n+1) = 0 \end{cases}$$

et celle du prédicat $PlusGrand$ définie sur l'ensembles $\mathbb{N}^2$

$$\chi_{PlusGrand}(x,y) = \begin{cases} 1 & \text{si} \quad x > y \\ 0 & \text{sinon.} \end{cases}$$

Définition 2.2.17 ***Prédicat primitif récursif***

Un prédicat P sur $\mathbb{N}^k$ est primitif récursif si sa fonction caractéristique χ_P est primitive récursive.

Un prédicat P est décidable si χ_P est récursive totale. Au cas contraire on dit que P est indécidable.

On peut voir que $\chi_P(x) = sg(x)$ qui est RP et χ_P est donc RP. Par conséquent, le prédicat P est RP.

Proposition 2.2.18

Soient P et Q deux prédicats RP, alors $P \wedge Q$, $P \vee Q$, $\rceil P$ et $P \Longrightarrow Q$ sont aussi RP.

Autrement dit, la classe de prédicats RP est fermée par les opérations booléennes.

Preuve. Nous prouvons d'abord que $P \wedge Q$ est RP.

Comme $(P \wedge Q)(x) \equiv P(x) \wedge Q(x)$, alors:

$$\chi_{(P\wedge Q)}(x) = \begin{cases} 1 & \text{si} \quad \chi_P(x) = 1 \ \text{ et } \chi_Q(x) \\ 0 & \text{sinon.} \end{cases}$$

On peut en déduire que $\chi_{(P\wedge Q)}(x) = \chi_P(x).\chi_Q(x)$, Or χ_P et χ_Q sont RP par hypothèse et, de plus, on sait que la multiplication est RP. Par conséquent, $\chi_{(P\wedge Q)}$ est RP et $P \wedge Q$ l'est aussi

Nous allons maintenant prouver que $\rceil P$ est RP.

$$\chi_{\rceil P}(x) = \begin{cases} 1 & \text{si} \quad \chi_P(x) = 0 \\ 0 & \text{si} \quad \chi_P(x) = 1 \end{cases}$$

Il est facile de voir que $\chi_{\rceil P}(x) = 1 - \chi_P(x)$. Alors, $\chi_{\rceil P}$ est RP et $\rceil P$ est donc RP

Pour "$\vee$" et "$\Rightarrow$", on peut les exprimer en utilisant "$\wedge$" et "$\rceil$" comme suit :

$$(P \vee Q) \equiv \rceil(\rceil P \wedge \rceil Q) \ \text{ et } \ (P \Rightarrow Q) \equiv (\rceil(P \wedge \rceil Q))$$

■

Si les fonctions g_i et les prédicats P_i sont RP (pour $i \in \{1, ..., n\}$, alors la fonction

$$f(x) = \begin{cases} g_1(x) & \text{si} \ \ P_1(x) \\ \vdots & \\ g_n(x) & \text{si} \ \ P_n(x) \end{cases}$$

est aussi RP.

Pour prouver la proposition, notons que la fonction f peut être écrite comme

$$f = g_1.\chi P_1 ... + g_n.\chi P_n$$

Alors, f est primitive récursive puisqu'elle est obtenue à partir des fonctions primitive récursive en utilisant l'addition et la multiplication.

2.2.2 Les foncrions μ récursives

Les fonctions μ récursives se définissent comme les fonctions primitives récursives mais permettent l'utilisation d'une nouvelle opération: la minimisation non bornée. Elle est semblable à la minimisation bornée, à cela prés qu'elle ne précise pas de borne. La minimisation non bornée d'un prédicat $P(\overline{n}, i)$ est une fonction que l'on note $\mu i\ P(\overline{n}, i)$ et on la défini comme suit:

Définition 2.2.19 ***minimisation non bornée***

Soit $P(\overline{n}, i)$ un prédicat. Alors

$$f(\overline{n}, i) = \mu i\ P(\overline{n}, i) = \begin{cases} le\ plus\ petit\ i \quad \text{telque} \quad P(\overline{n}, i) = 1 \quad \text{s'il existe} \\ 0 \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{si un tel } i \text{ n'existe pas} \end{cases}$$

On dit que f est obtenue de P par la minimisation non bornée

Si P est primitive récursive, et f est obtenue de P par la minimisation non bornée, alors f est aussi primitive récursive.

2.2.3 Fonctions récursives partielles

Nous avons vu que la classe des fonctions primitives récursives est en fait une sous-classe des fonctions calculables. La question qui se pose ensuite est : comment construire la vraie classe des fonctions calculables. En effet, soit $f_0(x), f_1(x), ..., f_i(x)...$ la liste complète des fonctions calculables telle que $f_i(x) = Int(i, x)$. Alors, $h(x) = f_x(x) + 1$ est une fonction calculable qui n'est pas dans la liste. La solution à ce problème est de considérer les fonctions partielles.

Définition 2.2.20 ***Fonction partielle***

Une fonction partielle $f : \mathbb{N}^k \to \mathbb{N}$ est une fonction d'un sous-ensemble de $\mathbb{N}^k$ vers $\mathbb{N}$. Le domaine de f est $Dom(f) = \{x / f(x) \text{ est définie}\}$.

Si en x la fonction f est définie, on écrit $f(x) \downarrow$ (on lit : f converge sur x), sinon on écrit $f(x) \uparrow$ (on lit : f diverge).

Si $Dom(f) = \mathbb{N}^k$ on dit que f est total.

2.2.4 Fonctions récursives partielles et la thèse de Church

Définition 2.2.21 ***Fonctions récursives partielles***

La classe des fonctions récursives partielles est la plus petite classe de fonctions partielles qui contient les fonctions de base O, σ et π, et qui est fermée par les constructeurs $Comp$, Rec et μ.

La seule différence par rapport à la définition de la classe RP est le nouveau constructeur μ. C'est trivial que toutes les fonctions récursives primitives RP sont aussi récursives partielles.

Un lecteur attentif peut remarquer ici, que pour donner un sens précis à cette définition, il faut étendre les opérations $Comp$, Rec aux fonctions partielles.

Thèse.1. (Thèse de Church)

La classe des fonctions récursives partielles est égale à la classe des fonctions calculables par un algorithme quelconque.

Notons que ceci est une thèse (pas un théorème) puisque la classe de fonctions récursives partielles est formellement définie tandis que "la classe des fonctions calculables" est une notion intuitive et informelle.

La thèse de Church contient deux parts :

1. Chaque fonction récursive partielle est calculable.
2. Chaque fonction calculable est récursive partielle.

2.3 Au-delà des fonctions primitives récursives

Toutes les fonctions primitives récursives sont calculables par une procédure effective. En effet, les fonctions de base sont calculables; la composition de fonctions calculables est calculable; de même la récursion primitive.

Il existe des fonctions qui ne sont pas priimitves récursives. En effet, il y a un nombre dénombrable de fonctions primitives récursives et un nombre non dénombrable de fonctions.

L'ensemble des fonctions primitives récursives est dénombrable, car chaque fonction primitive récursive peut être décrir par une chaîne de caractères.

Il existe des fonctions calculables qui ne sont pas primitives récursives.

Preuve. Nous utilisons le fait que les fonctions primitives récursives sont dénombrables. Soit $f_0, f_1, f_2, ...$une énumération de ces fonctions.

Considérons alors le tableau suivant :

$$\begin{array}{cccccccc}
A & 0 & 1 & 2 & \dots & j & \dots \\
f_0 & f_0(0) & f_0(1) & f_0(2) & \dots & f_0(j) & \dots \\
f_1 & f_1(0) & f_1(1) & f_1(2) & \dots & f_1(j) & \dots \\
f_2 & f_2(0) & f_2(1) & f_2(2) & \dots & f_2(j) & \dots \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \\
f_i & f_i(0) & f_i(1) & f_i(2) & \dots & f_i(j) & \dots \\
\vdots & \vdots & \vdots & \vdots & & \vdots & \ddots
\end{array}$$

Chaque case $A\,[i, j]$ de ce tableau contient un naturel qui est $f_i(j)$. utilisons-le pour définir une nouvelle fonction :

$$g(n) = f_n(n) + 1 = A\,[n, n] + 1$$

Cette fonction n'est pas primitive récursive. En effet, pour que ce soit le cas, il faudrait qu'elle soit identique à une des fonctions f_k que nous avons énumérées dans le tableau ci-dessus. Or cela est impossible car g aura une valeur différente de celle de f_k lorsque son argument vaut k puisque $g(k) = f_k(k) + 1$. la fonction g est néanmoins calculable. ■

Chapitre 3

Etude d'une fonction récursive non primitive récursive

Dans les années 1920, *Wilhelm Ackermann* et *Gabriel Sudan*, étudiaient les fondements de la calculabilité. *Sudan* est le premier à donner un exemple de fonction récursive mais non primitive récursive, appelée alors *fonction de Sudan*. Peu après et indépendamment, en 1928, *Ackermann* a publié son propre exemple de fonction récursive mais non primitive récursive. *Ackermann* considéra une fonction $A(m, n, p)$ dépendante de trois variables, et consistant à calculer la puissance itérée p fois de m par n, c'est-à-dire $m \to n \to p$ (dite la notation de *Conway)*.

Ackermann démontra que sa fonction A était bien une fonction récursive, i.e. une fonction qu'un ordinateur idéalisé peut calculer. Dans l'infini, *David Hilbert* conjectura que la fonction d'Ackermann n'était pas primitivement récursive. Cette conjecture fut établie par *Ackermann* lui-même dans son article *Zum Hilbertschen Aufbau der reellen Zahlen2*.

Pour plus de détails sur les résultas de ce chapitre on pourra consulter [12]

3.1 Fonction d'Ackermann

Définition 3.1.22

La fonction d'Ackermann notée $A(m, n)$ est une fonction totale, calculable mais non primitive récursive.

Cette fonction est définie comme suit :

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

Autrement dit :

$A(0,n) = n+1$

$A(m+1,0) = A(m,1)$

$A(m+1,n+1) = A(m, A(m+1,n))$

Cette définition n'a pas la forme d'une définition primitive récursive.

Ackermann a cependant lui-même initialement donné cette définition à trois paramètres

$$\begin{cases} \phi(m,n,0) = m+n \\ \phi(m,0,1) = 0 \\ \phi(m,0,2) = 1 \\ \phi(m,0,p+2) = m \\ \phi(m,n+1,p+1) = \phi(m,\phi(m,n,p+1),p) \end{cases}$$

Elle satisfait aux égalités suivantes :

$\phi(a,b,0) = a+b$

$\phi(a,b,1) = a.b$

$\phi(a,b,2) = a^b$

Table de valeurs

Valeurs de A(m,n)

$m\backslash n$	0	1	2	3	n
0	1	2	3	4	$n+1$
1	2	3	4	5	$n+2$
2	3	5	7	9	$2n+3$
3	5	13	29	61	$2^{n+3}-3$

Théorème 3.1.23 *la fonction d'Ackermann n'est pas récursivement primitive.*

Idées de la preuve.

a - On montre que les A_a les fonctions primitives récursives dans le sens suivant : pour toute f primitive récursive, il existe un entier a tel que f est majorée par A_a (à partir d'un certain rang).

Dans le cas où f a plusieurs variables, cela signifie que:

$f(a_1, ..., a_k) \leq A_a(max\{a_1, ...a_k\})$.

b - Diagonalisation : on considére $g : \mathbb{N} \to \mathbb{N}$ définie par: $g(a) = A_a(a, a) = A_a(a)$.

Parmi les propriétés élémentaires de A, on montre que $\forall a \in \mathbb{N}$, $A(a, a+1) < A(a+1, a+1)$. Ce qui donne $A_a(a+1) < g(a+1)$ et donc A_a ne majore pas g. On constate donc que g n'est pas primitive récursive puisqu'elle n'est pas bornée par une A_a.

Enfin, A n'est pas primitive récursive parceque, si elle l'était, g le serait aussi puisqu'elle s'obtient par composition à partir de A ($g = C(A, \pi_1^1, \pi_1^1)$).

Pour tout m et pour tout n, $A_m(n) > n$.

Par récurence sur m. Pour $m = 0$ et pour tout n, $A_0(n) = n + 1 > n$.

Supposons que, pour tout n on a $A_m(n) > n$ et vérifions que, pour tout n, $A_m+1(n) > n$.

On procède par récurence. sur n. Si $n = 0$, alors par hypothèse. de récurence. sur m, on a bien $A_{m+1}(0) = A_m(1) > 1 > 0$.

Supposons que $A_{m+1}(n) > n$ et vérifions que $A_{m+1}(n + 1) > n + 1$. Il vient en utilisant successivement les hypothèses de récurrence sur m et sur n $A_{m+1}(n + 1) = A_m(A_{m+1}(n)) > A_{m+1}(n) > n$, d'où le résultat annoncé, $A_{m+1}(n + 1) > n + 1$, puisque nous sommes en présence de deux inégalités strictes consécutives.

Pour tout m, la fonction A_m est strictement croissante, i.e., $A_m(n + 1) > A_m(n)$.

Si $m = 0$, c'est immédiat puisque $A_0(n) = n + 1$.

Si $m > 0$, il vient que $A_m(n + 1) = A_{m-1}(A_m(n)) > A_m(n)$.

Importance épistémologique

La *fonction d'Ackermann* croît extrêmement rapidement; $A(4, 2)$ a déjà 19729 chiffres, et représente bien plus que le nombre d'atomes estimé dans l'univers. Cette extrême croissance peut être exploitée pour montrer que la fonction f définie par $f(n) = A(n, n)$ croît plus rapidement que n'importe quelle fonction primitive récursive et ainsi que A n'est pas primitive récursive.

Cette fonction est néanmoins définissable par récursion primitive d'ordre supérieur. La fonction d'Ackermann constitue donc un exemple de fonction récursive, mais non primitive récursive. C'est peut-être l'exemple le plus célèbre de fonction récursive mais non primitive récursive, et c'est ce pour quoi elle est connue principalement, cependant son intérêt épistémologique va au delà.

L'exemple de la fonction d'Ackermann, montre que la notion de calculabilité introduite par les fonctions primitives récursives n'est pas la notion de calculabilité la plus générale, celle atteinte avec un grand succès par la thèse de Church. En effet, la fonction d'Ackermann est calculable au sens de *Turing* et de *Church*, mais pas à l'aide d'une fonction primitive récursive. C'est un exemple qui montre que *la* thèse de Church ne concerne pas tous les systèmes de calcul. Il existe des systèmes de calcul, qui semblent pourtant généralistes et puissants comme les fonctions primitives récursives, et qui effectivement permettent de définir la plupart des fonctions usuelles, mais qui ne sont pas équivalents aux machines de Turing. La calculabilité de la fonction d'Ackermann sert ici de critère distinctif, ceci révèle sa véritable importance épistémologique.

Intuitivement, la fonction d'Ackermann génère progressivement une multiplication par deux (additions réitérées), une exponentiation de base 2 (multiplications réitérées), une exponentiation réitérée, une réitération de cette opération, et ainsi de suite. Elle peut être exprimée en utilisant *la notation des puissances itérées de Knuth* :

$A(1,n) = 2 + (n+3) - 3$

$A(2,n) = 2 \times (n+3) - 3$

$A(3,n) = 2 \uparrow (n+3) - 3$

$A(4,n) = 2 \uparrow (2 \uparrow (2 \uparrow (2 \uparrow (n+3)))) - 3$

$\qquad = 2 \uparrow\uparrow (n+3) - 3$

$A(5,n) = 2 \uparrow\uparrow\uparrow (n+3) - 3$

On montre assez facilement par récurrence que:

$A(u,v) = 2 \uparrow (u-2)(v+3) - 3$

Réciproque

Puisque la fonction f définie par $f(n) = A(n,n)$ considérée précédemment croît réellement très vite, sa réciproque croît vraiment très lentement. Il est intéressant de remarquer

que cette réciproque apparaît dans l'analyse de la complexité de certains algorithmes, tels que l'algorithme Union-Find et un algorithme de calcul de l'arbre couvrant de poids minimal.

Applications pratiques.

La fonction d'Ackermann demandant beaucoup de calculs même pour de petites entrées, elle est parfois utilisée comme programme de test d'une implémentation d'un langage de programmation : en particulier, elle utilise de façon très exigeante la récursivité, de même que ses consœurs fib (suite de Fibonacci) et tak (fonction de Takeuchi).

Chapitre 4

Fonctions récursives et Machines de Turing

Nous allons maintenant donnés un autre modèle de calcul qui est *la machine de Turing*. Ce modèle est trés différent du modèle fonctionnel des fonctions récursives partielles. La raison pour laquelle nous concentrons sur les machines de Turing et qu'elles sont des machines abstraites trés simples mais capables de réaliser tous les algorithmes.

4.1 Définition

4.1.1 Description

Une machine de Turing déterministe est composée des éléments suivants:

- Une mémoire infinie sous forme de ruban divisé en cases. Chaque case du ruban peut contenir un symbole d'un alphabet de ruban.
- Une tête de lecture se déplaçant sur le ruban.
- un ensemble fini d'états parmi lesquels on distingue un état initial et un ensemble d'états accepteurs.
- Une fonction de transition qui pour chaque état de la machine et symbole se trouvant sous la tête de lecture précise

4.1.2 Formalisation

Une machine de Turing M est formellement décrite par un septuplet $(Q, \Gamma, \Sigma, \delta, s, B, F)$ tel que:

- Q est un ensemble fini d'états.
- Γ est l'alphabet de ruban (l'alphabet utilisé sur le ruban).
- $\Sigma \subset \Gamma$ est un alphabet d'entrée ne contient pas le symbole blanc (#).
- $s \in Q$ est l'état initial.
- $F \subseteq Q$ est l'ensemble des états accepteurs.
- $B \in \Gamma - \Sigma$ est le symbole blanc, que l'on notera (#).
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{G, D\}$ est la fonction de transition.

Il y a donc trois types d'instructions :

1. $(q, a) \to (p, b, G)$ (en observant a en état q passer à l'état p, écrire b, déplacer la tête à gauche),
2. $(q, a) \to (p, b, D)$ (déplacer à droite),
3. $(q, a) \to (p, b, -)$ (aucun déplacement).

4.1.3 Exécution

- Initialement, le mot d'entrée se trouve au début du ruban, Les autres cases du ruban contiennent toutes un symbole blanc partout ailleurs, La tête de lecture est sur la première case du ruban (extrémité gauche) et la machine se trouve dans son état initial.
- A chaque étape, la machine si la fonction de transition est définie :
  - lit le symbole se trouvant sous sa tête de lecture,
  - remplace ce symbole suivant par la fonction de transition,
  - déplace sa tête de lecture d'une case vers la gauche ou vers la droite suivant le sens précisé par la fonction de transition,
  - change d'état comme indiqué par la fonction de transition.

- Mot accepté par la machine lorsque l'exécution de celle-ci atteint un état accepteur, est rejeté si l'exécution s'arrête avant d'atteindre un état accepteur ou est infinie et n'atteint jamais d'état accepteur.

4.1.4 Configuration

Information nécessaire :

1 - l'état,

2 - le contenu du ruban,

3 - la position de la tête de lecture.

Remarque : à tout moment dans l'exécution, il existe une position sur le ruban à partir de laquelle il n'y a que des (#).

Représentation : triplet (q, α_1, α_2) contenant

1 - l'état de la machine ($q \in Q$).

2 - le mot sur le ruban avant la tête de lecture ($q \in \Gamma^*$).

3 - le mot sur le ruban après la tête ($\alpha_2 \in \{\varepsilon\} \cup \Gamma^*$).

Soit une configuration (q, α_1, α_2). Écrivons cette configuration sous la forme $(q, \alpha_1, b\alpha_2^{'})$ en prenant $b = \#$ dans le cas où $\alpha_2 = \varepsilon$. Les configuration dérivable à partir de (q, α_1, α_2) sont alors définies comme suit:

- Si $\delta(q, b) = (q^{'}, b^{'}, D)$ nous avons

$$(q, \alpha_1, b\alpha_2^{'}) \vdash_M (q^{'}, \alpha_1 b^{'}, \alpha_2^{'}).$$

- Si $\delta(q, b) = (q^{'}, b^{'}, G)$ et si $\alpha_1 \neq \varepsilon$ et est donc de la forme $\alpha_1^{'}a$ nous avons

$$(q, \alpha_1^{'}a, b\alpha_2^{'}) \vdash_M (q^{'}, \alpha_1^{'}, ab^{'}\alpha_2^{'}).$$

4.2 Langage accepté par une machine de Turing

Le langage accepté par une machine de Turing se définit à l'aide des notions de configuration et de dérivation entre configurations.

Le langage $L(M)$ accepté par une machine de Turing est l'ensemble des mots w tels que

$$(s, \varepsilon, w) \vdash_M^* (p, \alpha_1, \alpha_2), \text{ avec } p \in F.$$

Exemple 4.2.24

Soit la machine de Turing $M = (Q, \Gamma, \Sigma, \delta, s, B, F)$, où:

$Q = \{q_0, q_1, q_2, q_3, q_4\}$

$\Gamma = \{a, b, X, Y, \#\}$

$\Sigma = \{a, b\}$

$s = q_0$

$B = \#$

$F = \{q_4\}$

δ est donné par le tableau ci dessous (le symbole "$-$" indique que la fonction de transition n'est pas définie ces valeur).

	a	b	X	Y	$\#$
q_0	(q_1, X, D)	$-$	$-$	(q_3, Y, D)	$-$
q_1	(q_1, a, D)	(q_2, Y, D)	$-$	(q_1, Y, D)	$-$
q_2	(q_2, a, G)	$-$	(q_0, X, D)	(q_2, Y, G)	$-$
q_3	$-$	$-$	$-$	(q_3, Y, D)	$(q_4, \#, D)$
q_4	$-$	$-$	$-$	$-$	$-$

On peut se convaicre que cett machine de Turing accepte le langage $a^n b^n$ pour $n > 0$. En effet, elle remplace de façon répétée une paire de symbole a et b respectivement par Xet Y. À titre d'exemple la suite des configurations obtenues pour le mot d'entrée *aaabbb* est donnée comme suit:

$(q_0, \varepsilon, aaabbb)$	$(q_1, XXXYY, b)$
$(q_1, X, aabbb)$	$(q_2, XXXY, YY)$
$(q_1, Xa, abbb)$	(q_2, XXX, YYY)
(q_1, Xaa, bbb)	$(q_2, XX, XYYY)$
$(q_2, Xa, aYbb)$	(q_0, XXX, YYY)
$(q_2, X, aaYbb)$	$(q_3, XXXY, YY)$
$(q_2, \varepsilon, XaaYbb)$	$(q_3, XXXYY, Y)$
$(q_0, X, aaYbb)$	$(q_3, XXXYYY, \varepsilon)$
$(q_1, XX, aYbb)$	$(q_0, XXXYYY\#, \varepsilon)$

4.2.1 Fonctions calculées par une machine de Turing

Une machine de Turing peut calculer une fonction. L'argument de la fonction est le mot d'entrée et la valeur de la fonction est inscrite sur le ruban lorsque la machine de Turing a terminé son exécution.

Une machine de Turing calcule une fonction $f : \Sigma^* \longrightarrow \Sigma^*$ si, pour tout mot d'entrée w, elle séarrê toujours dans une configuration où $f(w)$ se trouve sur le ruban.

Une fonction est alors calculable par machine de Turing s'il existe une machine de Turing qui la calcule.

4.2.2 Thèse de Church-Turing. Équivalence M – fonctions récursives partielles

Nous présentons maintenant les résultats importants qui montrent le lien entre les fonctions calculées par les machines de Turing et les fonctions récursives partielles.

Théorème 4.2.25 *Chaque fonction récursive partielle peut être calculée par une machine de Turing.*

Théorème 4.2.26 *Chaque fonction calculée par une machine de Turing est récursive partielle.*

D'où la forme équivalente de *la thèse de Church.*

Thèse 2 (Thèse de Church-Turing)

La classe des fonctions calculables par machines de Turing est égale à la classe des fonctions calculables par un algorithme quelconque.

Conclusion

Dans ce mémoire, on a vue l'étude des fonctions récursives qui est une modèle de calcul construit à partir de trois fonctions de bases et les trois opérateurs: composition, récursion et minimisation.

Toute fonction calculable mécaniquement (à l'aide d'un algorithme) peut être obtenue par ces fonctions de bases et ces constructeurs.

On présente aussi un autre modèle plus puissant qui sont les machines de Turing.

Bibliographie

[1] B. Mariou. *Logique et complexité,* Cours university Paris 8, 2005.

[2] Ch. Boitet. *Récursivité, calculabilité, application à la théorie des langages*: *notes de cours -Grenoble* : Université scientifique et médicale, 1983, 1987.

[3] D. Davis, J. weyuker. *Computability, Complexity, and Languages* Academic Press, 1983.

[4] D. Lascar. *Logique mathématique*, tome 2. René Cori, 2011-2012.

[5] E. Asarin. *Calculabilité,* Cours université de Grenoble, 2003.

[6] G A. Giraldi. *Concepts in Theory of Computation and Applications for Automata-Based Modeling*, Petrópolis, RJ - Brasil, 2005.

[7] H. Rogers. *Theory of Recursive Functions and Effective Computability*, MIT Press, 1987.

[8] J. Hein. *Discrete Structures, Logic, and Computability,* Portland State University, 1995.

[9] J-P. Delahaye. *Calculabilité et décidabilité,* Cours université des Sciences et Technologies de Lille Laboratoire d'Informatique Fondamentale de Lille.

[10] M. Sipser. *Introduction to the Theory of Computation, Second Edition,* Thomson, 2006.

[11] P. Wolper. *Introduction à la calculabilité* : *Cours et exercices corrigés*, 2e édition, Dunod, 2001.

[12] http://fr.wikipedia.org/w/index.php?title=Fonction d'Ackermann&oldid=89770581.

Résumé

Ce travail constitue une introduction à la théorie de la calculabilité .

Le contenu est composé de quatres chapitres. Le premier «Notions sur la calculabilité» présente les notions généreaux de la calculabilité. Le deuxième «Fonctions et prédicats primitifs récursifs» donne des définitions formelle des fonctions et prédicats primitifs récursifs et on donnons quelque exemples.

Le troisième chapitre est un «Étude d'une fonction non primitive récursive» on particulions la fonction d'Ackremann.

Le dernier chapitre «Fonctions récursives et machines de Turing» on utilisons un autre modèle du calcul pour étudier les fonctions primitives récursives.

Abstract

This work provides an introduction to the theory of computability.

The content consists of four chapters. The first "Basics computability" presents the notions of computability-generous. The second "Functions and primitive recursive predicates" gives formal definitions of functions and primitive recursive predicates and give some examples.

The third chapter is a "Study of a non-primitive recursive function" based on the Ackremann partécullions.

The last chapter "recursive functions and Turing machines" we use another model of computation to study the primitive recursive functions.