Cours Logique et Calculabilité

L3 Informatique 2014/2015

Texte par Séverine Fratani, avec addenda par Luigi Santocanale
Version du 30 mars 2015

Table des matières

Chapitre 1

Calculabilité

Ce chapitre est largement inspiré du livre "Introductionà la calculabilité" de Pierre Wolper paru aux éditions Dunod [1].

1.1 Introduction

La question posée dans ce chapitre est de savoir quels problèmes sont solubles par un programme exécuté par un ordinateur.

Un *problème* est une fonction $f : I \to R$ d'un ensemble I d'instances vers un ensemble R de réponses possibles aux problème posé. Quand R est un ensemble binaire, on parle de *problème de décision* : la réponse attendue est *vrai* ou *faux*.

Quelques exemples de problèmes

1. Déterminer si un entier naturel est pair ou impair.
2. Trier un tableau de nombre.
3. Déterminer si un programme C s'arrête, quelles que soient les données fournies en entrée.
4. Déterminer si un polynôme à coefficients entiers a des racines entières (dixième problème de Hilbert).

On dira qu'un problème $f : I \to R$ est *calculable*, si il existe une procédure effective qui peut être appliquéeà n'importe instance $\iota \in I$ et qui a pour effet de produire le résultat $f(\iota)$.

Il se pose donc la question de définir la notion de *procédure effective* : elle doit être donnée par un ensemble fini d'instructions, utilisant des opérations très élémentaires, et doit pouvoir fournir le résultat en un nombre fini d'étapes. Par contre, on n'impose pas de limitation sur la taille de la mémoire disponible, ni sur la taille des données.

Cette présentation reste très informelle, il n'y a pas de définition rigoureuse de ce qu'est une procédure effective. Le but de ce cours est de présenter quelques *modèles* de cette notion en s'appuyant sur la thèse de Church-Turing.

Thèse de Church-Turing

1. Aucun modèle de la notion de fonction calculable n'est plus puissant que les machines de Turing.
2. Toute fonction calculable (au sens intuitif) est calculable par une machine de Turing.
3. Toutes les définitions formelles de calculabilité connuesà ce jour sont équivalentes.
4. Toutes les formalisations de la calculabilité établies par la suite seront équivalentes aux définitions connues.

Cette thèse est universellement reconnue comme vraie, bien que seul le troisième point peut être démontré.

Actuellement, on considère que les fonctions calculables sont celles calculées par un programme (peu importe le langage choisi).

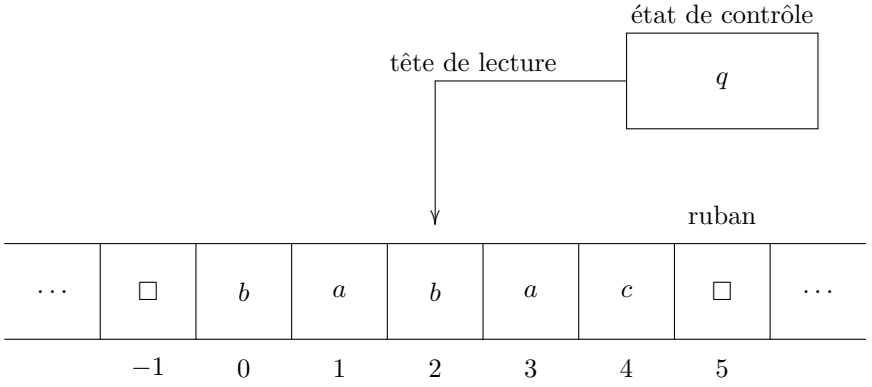

FIGURE 1.1 – Machine de Turing, intuitions

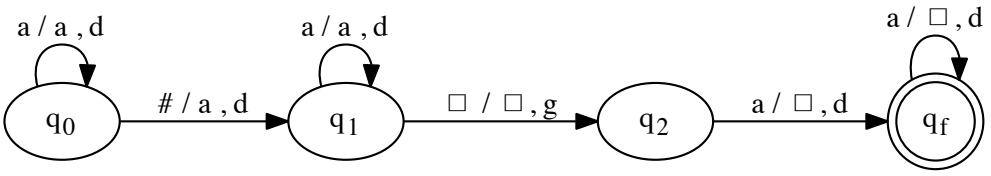

FIGURE 1.2 – Machine de Turing pour la somme

Nous présenterons dans ce cours deux notions de fonctions calculables : les fonctions calculables par une machines de Turing, les fonctions μ-calculables.

1.2 Machines de Turing

Définition 1.1. Une **machine de Turing** est une structure $T = \langle Q, \Gamma, \Sigma, \delta, q_0, B, F \rangle$, où
- Q est un ensemble fini d'*états* ;
- Γ est l'*alphabet de ruban* ;
- $\Sigma \subseteq \Gamma$ est l'*alphabet d'entrée* ;
- $q_0 \in Q$ est l'*état initial* ;
- $F \subseteq Q$ est l'ensemble des *états accepteurs* (ou finaux) ;
- $B \in \Gamma - \Sigma$ est le symbole blanc, souvent noté $\square$;
- $\delta : (Q - F) \times \Gamma \to Q \times \Gamma \times \{ -1, +1 \}$ est la *fonction* (partielle) *de transition.*

Remarque 1.2. On peut lire la fonction de transition

$$\Delta(q, \sigma) = (q', \sigma', d)$$

(avec $d \in \{ -1, 1 \}$) par : *si on se trouve dans l'état q et la tête de lecture lit la lettre σ, alors on remplace σ par σ', on rentre dans l'état q', et on déplace la tête sur le ruban en direction d* (c'est-à-dire,à gauche si $d = -1$,à droite si $d = +1$).

Le ruban est comme un grand tableau indexé par les entiers, dont les cellules contiennent des lettres de Γ. On peut donc décrire l'état du ruban par une fonction $r : \mathbb{Z} \longrightarrow \Gamma$. Par ailleurs, on considère qu'à tout moment l'information dans le ruban soit finie, c'est-à-dire que tout l'ensemble des cases qui ne contiennent pas le symbole $\square$ soit fini.

A tout moment d'un calcul, l'état global de la machine peut se décrire par le contenu du ruban, la position de la tête de lecture sur le ruban, et l'état de contrôle (voir la Figure 1.1). On peut formaliser la notion d'état global par la définition suivante, où "état global" devient "configuration".

Définition 1.3. Une **configuration** de T est un triplet (q, u, v) où :
- $q \in Q$ est l'état de contrôle ;
- $u \in \Gamma^*$ est le mot apparaîssant avant la position de la tête de lecture ;
- $v \in \Gamma^*$ est le mot apparaissant sur le ruban entre la position de la tête de lecture et le dernier caractère non blanc. Ce mot est donc fini et ne termine pas par $\square$.

Les configurations initiales sont toutes les configurations $c_\iota(w) = (q_0, \varepsilon, w)$, $w \in \Sigma^*$ Une *configuration* (q, u, v) est *finale* si $q \in F$.

Etant donnée une configuration (q, u, v), on peut toujours trouver $b \in \Gamma$ et $v' \in \Gamma^*$ tel que $v = bv'$ (Si $v = \varepsilon$, on pose $v' = \varepsilon$ et $b = \square$). De même, on pourra supposer que u se décompose en $u = u'b$.

Définition 1.4. Les **configurations dérivables** à partir de (q, u, v) sont définies de la façon suivante :
- si $\delta(q, b) = (q', a, +1)$, alors
$$(q, u, bw') \vdash_T (q', ua, w') ;$$
- si $\delta(q, b) = (q', a, -1)$, alors
$$(q, u'c, bw) \vdash_T (q', u', caw) .$$

On pose $(q, u, w) \vdash_T^* (q', u', w')$ s'il existe une suite de configurations $(q, u, w) = c_0, \ldots, c_n = (q', u', w')$ telle que $c_i \vdash_T c_{i+1}$, pour $i = 0, \ldots, n-1$.

Remarquez que, vue la définition donnée, une machine de Turing est déterministe : pour toute configuration c, il existe toujours au plus une configuration c' telle que $c \vdash_T c'$. De plus, une fois arrivée sur une configuration finale, la machine s'arrête.

Définition 1.5. On dit que T **s'arrête sur entrée** w (et on écrit cela par $T(w)\downarrow$) s'il existe une configuration finale c_f telle que $(q_0, \varepsilon, w) \vdash_T^* c_f$. Sinon, on dit que T **diverge sur entrée** w (et on écrit cela par $T(w)\uparrow$).

Langage reconnu par une machine de Turing. Le langage accepté par une machine de Turing est l'ensemble des mots $w \in \Sigma^*$ tels que

$$(q_0, \varepsilon, w) \vdash_T^* c_f \text{ où } c_f \text{ est une configuration finale.}$$

Le langage reconnu par une machine de Turing n'est pas reconnu par une procédure effective, en effet puisqu'on n'a pas interdit les calculs infinis, on ne peut pas toujours savoir si un mot est accepté ou pas.

Langage décidé par une machine de Turing. Un langage L est décidé par une machine de Turing T si T accepte L et T ne contient pas d'exécutions infinies.

Le langage décidé par une machine de Turing peut donc être reconnu par une procédure effective.

Définition 1.6. Un langage est dit **récursif** si il est décidé par une machine de Turing. Il est dit **récursivement énumérable** si il est accepté par une machine de Turing.

Fonction calculée par une machine de Turing.

Définition 1.7. Une machine de Turing calcule une fonction $f : \Sigma^* \to \Gamma^*$ si pour tout mot d'entrée w, elle s'arrête dans une configuration finale où $f(w)$ se trouve sur le ruban.

Une fonction est **calculable par une machine de Turing** si il existe une machine de Turing qui la calcule.

Exercice 1.8. Exécutez la machine de Turing de la figure 1.2 à partir de la configuration initiale $(q_0, \varepsilon, aa\sharp aaa)$. Que calcule, en général, cette machine ?

1.3 Problèmes de décision

On rappelle qu'un *problème de décision* est une fonction $P : I_P \to \{ Oui, Non \}$. On dit que I_P est l'ensemble des *instances de* P.

Exemple 1.9. SAT est le problème de décision suivante : une instance de SAT (donc un élément de $I_{\texttt{SAT}}$) est un ensemble fini de clauses ; pour un tel ensemble de clauses $\mathcal{C}$, on aura $\texttt{SAT}(\mathcal{C}) = Oui$ si $\mathcal{C}$ est satisfaisable, et $\texttt{SAT}(\mathcal{C}) = NON$ sinon.

En principe, tous les problèmes peuvent se réduireà des problèmes de décision. Considérons, par exemple, le problème de trouver le nombre chromatique d'un graphe. Soit C_n le problème de décision, dont les instances sont les graphes non-orientés, et tel que $C_n(V, E) = Oui$ ssi (V, E) possède un coloriage avec n couleur. Pour trouver le nombre chromatique d'un graphe, nous pouvons résoudre les problèmes $C_1, \dots C_k \dots$ jusqu'à trouver une réponse Oui.

Un *codage* est une fonction injective qui permet de représenter les instances d'un problème comme des mots sur l'alphabet Σ d'une machine de Turing T. Nous allons supposer qu'il existe toujours un codage canonique des l'ensemble de réponses $\{ Oui, Non \}$ sur le langage d'une machine de Turing ; par conséquent, nous ne ferons pas mention de pas ce codage dans la suite.

Définition 1.10. Un problème de décision P est **décidable** s'il existe une machine de Turing T et un codage $\gamma : I_P \to \Sigma^*$ telle que :

1. $T(w)\downarrow$, pour tout $w \in \Sigma^*$;
2. pour toute instance $i \in I_P$, $T(\gamma(i)) = Oui$ ssi $P(i) = Oui$.

Attention, dans la définition précédente on ne peut pas omettre la première clause. Si on le fait, on obtient la définition de problème semi-décidable :

Définition 1.11. Un problème de décision P est **semi-décidable** s'il existe une machine de Turing T et un codage $\gamma : I_P \to \Sigma^*$ tels que, pour toute instance $i \in I_P$: $P(i) = Oui$ ssi $T(\gamma(i))\downarrow$ et $T(\gamma(i)) = Oui$.

La notion de semi-décidabilité est très différente, elle ne dit rien sur l'arrêt de la machine pour une instance i telle $P(i) = Non$: en fait, dans ce cas, on pourrait avoir $T(\gamma(i))\downarrow$ *ou bien* $T(\gamma(i))\uparrow$.

On peut aisément se convaincre de la caractérisation suivante :

Proposition 1.12. *Un problème de décision P est semi-décidable s'il existe une machine de Turing T et un codage $\gamma : I_P \to \Sigma^*$ tels que, pour toute instance $i \in I_P$,*

- *si $P(i) = Oui$ alors $T(\gamma(i))\downarrow$,*
- *si $P(i) = Non$ alors $T(\gamma(i))\uparrow$.*

Étant donné P, soit $\neg P$ le problème avec les mêmes instances de P tel que $(\neg P)(i) = Oui$ ssi $P(i) = Non$. On peut montrer la proposition suivante :

Proposition 1.13. *Si P et $\neg P$ sont semi-décidables, alors P est décidable.*

Observez que l'implication inverse, si P est décidable, alors P e $\neg P$ sont semi-décidables, est trivialement vraie.

1.4 Un problème indécidable

Le problème de l'ARRÊT a comme instances les couples (T, w) avec T une machine de Turing et w un mot sur l'alphabet Σ de la machine. $\texttt{ARRÊT}(T, w) = OUI$ si et seulement si la machine de Turing T s'arrête sur entrée w (ce que nous avons noté par $T(w)\downarrow$).

Proposition 1.14. *Le problème de l'ARRÊT n'est pas décidable.*

Démonstration. Par l'absurde : soit $\mathcal{N}$ une machine de Turing qui s'arrête toujours, et soit γ un codage, tels que, pour tout (T, w), on ait

$$T(w)\downarrow \quad \text{ssi} \quad \mathcal{N}(\gamma(T,w)) = Oui\,.$$

Soit Σ l'alphabet de $\mathcal{N}$; nous pouvons supposer que $\gamma(T,w)$ est de la forme $\alpha(T)\sharp\beta(w)$, où α est un codage des machines de Turing vers les mots sur l'alphabet $\Sigma \setminus \{\,\sharp\,\}$, et β est un codage des mots sur un langage avec un nombre arbitraire de symboles vers les mots sur $\Sigma \setminus \{\,\sharp\,\}$. On a donc :

$$T(w)\downarrow \quad \text{ssi} \quad \mathcal{N}(\alpha(T)\sharp\beta(w)) = Oui\,.$$

Construisons une machine de Turing $\mathcal{D}$ comme suit. $\mathcal{D}$ prend un mot w sur le ruban et il ajouteà la droite du mot w son codage $\beta(w)$, separé par le symbole $\sharp$. Le mot $w\sharp\beta(w)$ se trouveraà ce point sur le ruban. Ensuite $\mathcal{D}$ utilise $\mathcal{N}$ comme un sous-module, avec $w\sharp\beta(w)$ en entrée. Si $\mathcal{N}$ réponds Oui, alors $\mathcal{D}$ rentre dans une boucle infinie et ne s'arrête pas. Si $\mathcal{N}$ réponds Non, alors $\mathcal{D}$ rentre dans l'état final et s'arrête.

Maintenant, posons la question si $\mathcal{D}$, avec entrée $\alpha(\mathcal{D})$, s'arrête.

Si c'est le cas, c'est parce que $\mathcal{N}$, avec entrée $\alpha(\mathcal{D})\sharp\beta(\alpha(\mathcal{D}))$ a répondu Non ; par les hypothèses faites sur $\mathcal{N}$, cela est équivalentà dire que $\mathcal{D}$ avec entrée $\gamma(\mathcal{D})$ ne s'arrête pas (on a donc obtenu une contradiction).

Par ailleurs, si c'est ne pas le cas, c'est parce que $\mathcal{N}$, avec entrée $\alpha(\mathcal{D})\sharp\beta(\alpha(\mathcal{D}))$ a répondu Oui ; par les hypothèses sur $\mathcal{N}$, cela est équivalentà dire que $\mathcal{D}$ avec entrée $\alpha(\mathcal{D})$ s'arrête (on a donc une autre contradiction). □

Remarque 1.15. Que l'encodage $\gamma(T,w)$ soit de la forme $\alpha(T)\sharp\beta(w)$ est un raccourcis pour ne pas alourdir la preuve par des détails techniques, qui sont par ailleurs nécessaires. Soit $\mathcal{MDT}$ l'ensemble des Machine de Turing ; nous supposons en fait que :

1. il existe un alphabet fini Σ_0 tel que $\sharp \notin \Sigma_0$;
2. nous disposons d'un encodage canonique $\alpha : \mathcal{MDT} \to \Sigma_0^*$,
3. nous disposons un encodage canonique $\gamma : \mathbb{N}^* \to \Sigma_0^*$;
4. l'alphabet de n'importe quelle machine de Turing est de la forme $\Sigma = \{\,0, \ldots, n\,\}$ pour un quelque $n \geq 0$;
5. si P est un problème de décision tel que $I_P \subseteq \Sigma^*$, alors on a pas besoin d'encoder les instances de P comme des mots sur un autre alphabet.

Clairement, les hypothèses (1)à (4) ne posent pas de problèmes (via un renommage du symbol $\sharp$). L'hypothèse (5) peut être affaiblie, en demandant que si $I_P \subseteq \Sigma^*$, alors encodage de I_P soit calculé par une machine de Turing.

Étant dit cela et en dénotant par Σ_T l'alphabet d'une machine de Turing T, la façon correcte de définir le problème de l'arrêt est la suivante :

$$I_{\texttt{ARRÊT}} = \{\,\alpha(T)\sharp\beta(w) \in (\Sigma_0 \cup \{\,\sharp\,\})^* \mid T \text{ une machine de Turing, } w \in \Sigma_T^*\,\}\,,$$
$$\texttt{ARRÊT}(\alpha(T)\sharp\beta(w)) = Oui \text{ ssi } T(w)\downarrow\ .$$

Remarquons en autre que le probleme de l'arrêt est semi-décidable. En fait, le Théorème suivant établie qu'il existe une machine de Turing U (car elle est dite universelle), qui se comporte comme un système d'exploitation : étant donnée en entrée la description d'une machine de Turing, elle simule la machine donnée en entrée, et va produire comme résultat un codage du résultat.

Théorème 1.16. *Il existe une machine de Turing U, telle que, pour tout toute entrée de la forme $\alpha(T)\sharp\beta(w)$, on a :*

1. *$T(w)\downarrow$ ssi $U(\alpha(T)\sharp\beta(w))\downarrow$;*
2. *si $T(w)\downarrow$, alors $U(\alpha(T)\sharp\beta(w)) = \beta(T(w))$.*

Corollaire 1.17. *Le problème de l'arrêt est semi-décidable.*

Car il suffit de construire une machine A, qui utilise la machine U comme un sous-routine : dès que U rentre dans un état final (et donc elle retourne en tant que sous-routine), A remplace le contenu du ruban par le codage de la réponse positive Oui.

1.5 Fonctions récursives

La classe des fonction récursive est une classe de fonctions partielles (possiblement totales) de la forme $f : \mathbb{N}^n \to \mathbb{N}^m$, définie comme la plus petite classe de fonctions (partielles) contenant certaines fonction élémentaires et fermée sous certains schémas de définition.

Nous allons présenter ces schémas. En définissant la valeur $f(x)$ d'une fonction partielle, nous pouvons utiliser d'autres fonctions partielles qui peuvent ne pas être eux-mêmes définie sur ces valeurs. Si c'est le cas, alors il est implicite que la valeur de $f(x)$ n'est pas défini. Si nous souhaitons dire explicitement que la valeur de $f(x)$ n'est pas défini, alors nous allons écrire $f(x) = \perp$.

1.5.1 Les fonctions primitives recursives

Les fonctions primitives récursives sont définiesà partir de fonctions de base, d'une règle de composition, et d'une règle de récursion.

Les fonction de base sont les suivantes :

— La fonction $\mathbf{0}()$ qui n'a pas d'argument et retourne toujours 0.

— Les fonctions de projection :

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

qui permettent de sélection un argument parmi k.

— La fonction successeur :

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

On utilise aussi les schémas suivants :

Schéma de composition. Soit $f : \mathbb{N}^n \to \mathbb{N}^m$ et $g : \mathbb{N}^n \to \mathbb{N}^k$. Posons

$$h(x) = g(f(x)) .$$

On dit que $h : \mathbb{N}^n \to \mathbb{N}^k$ est définie par compositionà partir de f et g.

Schéma de duplication. Etant donné $f_i : \mathbb{N}^n \to \mathbb{N}^{n_i}$ avec $i = 1, \ldots, k$, posons

$$h(x) = (f_1(x), \ldots, f_n(x))$$

La fonction $h : \mathbb{N}^n \to \mathbb{N}^m$, avec $m = \sum_{i=1,\ldots,k} n_i$ est dite définie par duplication depuis les f_i.

Exemple 1.18. Si $f_1 = f_2$ est la fonction identité, alors la fonction diagonale $\Delta(x) = (x, x)$, qui duplique la valeur de x, est définie par duplication depuis f_1 et f_2.

Schéma de récursion primitive Soient $f : \mathbb{N}^{1+m} \to \mathbb{N}^m$ et $g : \mathbb{N}^n \to \mathbb{N}^m$. Posons :

$$h(0, \bar{y}) = g(\bar{y}) , \qquad h(x + 1, \bar{y}) = f(x, h(x, \bar{y})) .$$

On dit que $h : \mathbb{N}^{1+n} \to \mathbb{N}^m$ est définie par récursion primitive depuis f et g.

Définition 1.19. Les fonctions primitives récursives sont toutes les fonction de base, et toutes les fonctions obtenues à partir de fonctions primitives récursives de base par l'application des schémas de composition, de duplication, et de récursion primitive.

Les prédicats primitives récursifs sont les fonctions primitives récursivesà valeur dans $\{0, 1\}$.

1.5.1.1 Exemples

Constantes Toutes les fonctions constantes sont primitives récursives :

$$\mathbf{j}() = \sigma^j(\mathbf{0}())$$

Addition

$$\begin{aligned} plus(n_1, 0) &= n_1 \\ plus(n_1, n_2 + 1) &= \sigma(plus(n_1, n_2)) \end{aligned}$$

Produit

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

Prédecesseur

$$\begin{aligned} pred(m) &= 0 \text{ si } m = 0 \\ pred(m) &= m-1 \text{ si } m > 0 \end{aligned}$$

est primitive récursive puisqu'elle peut s'écrire par récursion primitive :

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

Soustraction

$$\begin{aligned} moins(n,m) &= 0 \text{ si } n \leq m \\ moins(n,m) &= n-m \text{ sinon} \end{aligned}$$

est primitive récursive puisqu'elle peut s'écrire par réxcursion primitive :

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

Signe

$$\begin{aligned} sg(n) &= 0 \text{ si } n = 0 \\ sg(n) &= 1 \text{ sinon} \end{aligned}$$

est primitive récursive puisqu'elle peut s'écrire par récursion primitive :

$$\begin{aligned} sg(0) &= 0 \\ sg(n+1) &= 1 \end{aligned}$$

Comparaisons Les fonctions caractéristiques des prédicats $<$, $=$, $\leq$ sont également primitifs récursifs :

$$\begin{aligned} ppetit(n,m) &= 1 \text{ si } n < m, 0 \text{ sinon} \\ egal(n,m) &= 1 \text{ si } n = m, 0 \text{ sinon} \end{aligned}$$

est primitive récursive puisqu'elle peut s'écrire par récursion primitive :

$$\begin{aligned} ppetit(n,m) &= sg(moins(m,n)) \\ egal(n,m) &= moins(1, sg(m-n) + sg(n-m)) \end{aligned}$$

La quantification bornée Si p est un prédicat primitif récursif, alors

$$\forall i \leq m, p(\bar{n}, i)$$
$$\forall i \leq m, p(\bar{n}, i)$$

sont des prédicats primitifs récursifs :

$$\begin{aligned} \forall i \leq m, p(\bar{n}, i) &= \Pi_{i=0}^{m} p(\bar{n}, i) \\ \exists i \leq m, p(\bar{n}, i) &= sg(\Sigma_{i=0}^{m} p(\bar{n}, i)) \end{aligned}$$

Fonction conditionnelle Soit la fonction définie par :

$$\begin{aligned} f(\bar{n}) &= g_1(\bar{n}) \text{ si } p_1(\bar{n}) \\ &= ... \\ &= g_\ell(\bar{n}) \text{ si } p_\ell(\bar{n}) \end{aligned}$$

Si les f_i sont des fonctions primitives récursives et les p_i sont des prédicats primitifs récursifs alors f est une fonction primitive récursive :

$$f(\bar{n}) = \Sigma_{i=1}^{\ell} g_i(\bar{n}) \times p_i(\bar{n})$$

La minimisation bornée Si p est un prédicat primitif récursif, alors

$$\begin{aligned} \mu i \leq m, p(\bar{n}, i) &= \text{ le plus petit } i \text{ tel que } p(\bar{n}, i) = 1 \\ &= 0 \text{ siun tel } i \text{ n'existe pas} \end{aligned}$$

est une fonction primitive récursive :

$$\begin{aligned} \mu i \leq 0, p(\bar{n}, i) &= 0 \\ \mu i \leq (m+1), p(\bar{n}, i) &= \mu i \leq m, p(\bar{n}, i) \text{ si } p(\bar{n}, m) = 0 \\ &= m+1 \text{ si } p(\bar{n}, m+1) \text{ et } \neg\exists i \leq m+1, p(\bar{n}, i) \end{aligned}$$

1.5.1.2 Comparaison avec les fonctions calculables

Les fonctions primitives récursives sont toutes calculables par une procédure effective (au sens de la thèse de Turing-Church). Il existe des fonctions calculables qui ne sont pas primitives récursives. En effet, l'ensemble des fonctions primitives récursives est dénombrable, puisque chaque fonction peut être décrite par une cabine de caractères (sa description en fonctions de base et opérations)

Théorème 1.20. *Il existe des fonctions calculables qui ne sont pas primitives récursives.*

Démonstration. Les fonctions primitives récursives sont dénombrables. Soit donc $f_0, f_1, \ldots, f_n, \ldots$ une énumération des fonctions primitives récursives.

Nous utiliserons la notation suivante pour simplifier : $f_i(n) = f_i(n, \ldots, n)$. On considère le tableau suivant :

A	0	1	2	...	j	...
f_0	$f_0(0)$	$f_0(1)$	$f_0(2)$	...	$f_0(j)$	...
f_1	$f_1(0)$	$f_0(1)$	$f_1(2)$	...	$f_1(j)$	...
f_2	$f_2(0)$	$f_2(1)$	$f_2(2)$	...	$f_2(j)$	...
$\vdots$						
f_i	$f_i(0)$	$f_i(1)$	$f_i(2)$	...	$f_i(j)$	...
$\vdots$						

La case $A[i,j]$ du tableau contient donc l'entier naturel $f_i(j)$.
On définit maintenant la fonctionà 1 argument suivante :

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

Clairement, cette fonction n'est pas primitive récursive car elle ne peut être égaleà aucune des f_i. Par contre elle est calculable. Pour calculer $g(n)$, on procède de la faÃ§on suivante :

1. On énumère les fonctions primitives récursives jusqu'à f_n. Cette énumération peut se faire par un algorithme qui énumère systématiquement toutes les chaines de caractères et conservent celles qui décrivent une fonction primitive récursive.
2. ensuite on évalue $f_n(n)$, qui est calculable car f_n est primitive récursive
3. On évalue $f_n(n) + 1$

□

Un exemple célèbre de fonction calculable non primitive récursive est la fonction d'Ackermann :

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

1.5.2 Fonctions μ-récursives

On les obtient comme les fonctions primitives récursives mais on autorise une opération supplémentaire : la minimisation non bornée.

La minimisation non bornée d'un prédicat $p(\bar{n}, i)$ généralise la minimisation bornée vue précédemment.

$$\begin{aligned} \mu i\, p(\bar{n}, i) &= \text{ le plus petit } i \text{ tel que } p(\bar{n}, i) = 1 \\ &= 0 \text{ si un tel } i \text{ n'existe pas} \end{aligned}$$

La minimisation non bornée appliquéeà des prédicats primitifs récursifs ne produit pas que des fonctions calculables. En effet, prenons un prédicat $p(\bar{n}, i)$ et un $\bar{n}$ tel que $p(\bar{n}, i) = 0$ pour tout i. Il va falloir tester tous les i pour évaluer $\mu i\, p(\bar{n}, i)$, et donc la procédure ne terminera jamais. Il faut donc se limiter aux prédicats "sûrs" qui vont assurer que la fonction sera calculable :

Définition 1.21. Un prédicat $p(\bar{n}, i)$ est dit sûr si

$$\forall \bar{n}, \exists i\, p(\bar{n}, i).$$

Définition 1.22. Les fonctions et prédicats μ-recursifs sont ceux obtenusà partir des fonctions primitives récursives de base par
- composition
- duplication
- récursion primitive
- minimisation non bornée de prédicats sûrs.

Cette définition assure que les fonctions μ-récursives sont calculables. Remarquons qu'il n'existe pas de procédure effective pour décider qu'un prédicat est sûr.

Définition 1.23. La classe de fonction μ-récursives est la plus petite classe $\mathcal{R}$ de fonctions partielles telle que :
- toute fonction constante appartientà $\mathcal{R}$;
- toute projection appartientà $\mathcal{R}$;
- la fonction successeur s (avec $s(x) = x + 1$) appartient à $\mathcal{R}$;
- $\mathbb{R}$ sous le schémas de duplication, récursion primitive, minimisation.

Théorème 1.24. *Une fonction est calculable par une machine de Turing ssi elle est une fonction μ-récursive.*

Bibliographie

[1] Pierre Wolper. *Introduction à la calculabilité*. Sciences sup. Dunod, Paris, France, troisième edition, October 2006.