Avertissement liminaire

Le contenu du polycopié

Au cœur de l'informatique, on rencontre la notion d'algorithme. Il est donc important de préciser les conditions dans lesquelles un problème peut être "résolu par un algorithme". C'est ce qu'on s'efforce de faire dans une des plus anciennes branches de l'informatique appelée **théorie de la calculabilité**[1]. Il vaut la peine de souligner d'emblée qu'on ne s'intéresse qu'à la possibilité "abstraite" de traiter algorithmiquement un problème ; dans le monde réel des informaticiens, l'exécution de certains algorithmes demande un temps de calcul rédhibitoire !

On trouvera dans les pages qui suivent une (assez brève) présentation des fondations de la théorie de la calculabilité. On a fait le choix de deux modèles de calcul : les programmes avec branchements (chapitre 1) et les fonctions récursives (chapitre 2). Le chapitre 3 est centré sur deux résultats majeurs, un théorème d'indécidabilité et le théorème d'universalité ; le lecteur mesurera son degré d'assimilation du cours à sa compréhension de l'un et l'autre théorèmes.

Le lecteur trouvera à la fin du volume une très courte bibliographie. On s'est délibérément limité aux ouvrages qui ont été utilisés pour la rédaction de ces notes. On recommande aux lecteurs de commencer par les quatre premiers chapitres du livre de Davis *et alii*, qui couvrent tous les thèmes abordés au cours des leçons orales. Les livres de Cori et Lascar (Tome II, chapitre 5), de Smith et de Stern sont à réserver pour un travail approfondi. Tous ces auteurs ont des objectifs bien plus ambitieux que le(s) rédacteur(s) de ces lignes.

[1]Tout est relatif : la théorie de la calculabilité n'a qu'une soixantaine d'années, selon l'estimation la plus généreuse !

On s'est efforcé de donner une démonstration pour chacun (ou presque!) des énoncés du cours. Toutefois, certaines démonstrations sont détaillées, d'autres à peine esquissées. En général, on a complètement rédigé celles qui constituent la première occurrence dans le texte d'une technique particulière. Quant aux démonstrations qui font appel à des méthodes de codages (un peu plus) subtiles, on les a rejetées en annexe. On conseille de ne s'y frotter qu'après avoir médité les trois premiers chapitres.

Notation des fonctions

On limite notre champ d'étude aux fonctions à valeurs entières. Soit p un entier naturel ≥ 1. Une application (ou fonction totale) f de $\mathbb{N}^p$ dans $\mathbb{N}$ associe à chaque p-uplet[2] d'entiers naturels $(a_1, \ldots, a_p)$ un unique entier naturel $f(a_1, \ldots, a_p)$. Une fonction partielle g de $\mathbb{N}^p$ dans $\mathbb{N}$ est une application d'un sous-ensemble E de $\mathbb{N}^p$ dans $\mathbb{N}$. E est appelé l'ensemble de définition de g ; on écrit aussi $E = \text{domaine}(g)$. Si $(a_1, \ldots, a_p)$ est un p-uplet qui n'appartient pas à domaine(g), on dit que $g(a_1, \ldots, a_p)$ n'est pas défini et on écrit $g(a_1, \ldots, a_p) \uparrow$; si $(a_1, \ldots, a_p) \in \text{domaine}(g)$, on dit que $g(a_1, \ldots, a_p)$ est défini et on écrit parfois $g(a_1, \ldots, a_p) \downarrow$

Si $p = 0$, $\mathbb{N}^p$ ne contient que la suite vide, et on convient d'identifier les applications[3] et l'ensemble $\mathbb{N}$: un entier naturel n peut donc être vu comme l'image d'une fonction f qui ne prend pas d'argument ; on notera $F :\to \mathbb{N}$ et $f() = n$.

Si p est un entier naturel, on note $\mathcal{F}_p$ l'ensemble des fonctions partielles de $\mathbb{N}^p$ dans $\mathbb{N}$. Par abus de langage, le mot "fonction" sera utilisé comme synonyme de "fonction partielle". Une fonction de $\mathcal{F}_n$ est dite n-aire (ou d'arité n) ; pour $n = 1, 2, 3$, on parle de fonctions respectivement unaires, binaires et ternaires.

[2] Un p-uplet est une suite de longueur p.

[3] et les fonctions partielles, car dans le cas présent, il n'y a plus de distinction à faire entre ces deux notions !

Table des matières

Chapitre 1

Programmes avec branchements

Dans ce premier chapitre, on définit une classe de programmes, appelés programmes avec branchements, qui constitue le langage de programmes $\mathcal{S}$. Puis on précise ce qu'on entend par les expressions "exécuter un programme" et "la fonction f est calculée par le programme $\mathcal{P}$". Enfin, on définit la classe des fonctions calculables comme la classe des fonctions calculées par un programme avec branchements. La dernière section porte sur l'utilisation des macros, qui sont des pseudo-instructions et permettent d'abréger l'écriture des programmes.

1.1 Les programmes avec branchements

1.1.1 La syntaxe du langage $\mathcal{S}$

Dans ce paragraphe, on précise quelles suites de symboles sont considérées comme des programmes du langage $\mathcal{S}$, sans se soucier du sens qu'on veut leur donner.

Définition 1.1 *Deux types de symboles joueront un rôle particulier dans les programmes : les* **variables** *et les* **étiquettes**.

– *On distingue trois types de variables :*

1. *Les* **variables d'entrée** *sont les symboles* $X_1, X_2, X_3, \ldots$
2. *La* **variable de sortie** *est le symbole* Y

3. *Les* **variables auxiliaires** *sont les symboles* $Z_1, Z_2, Z_3, \ldots$

– *Les étiquettes sont les symboles* $A_1, B_1, C_1, D_1, E_1, A_2, B_2, C_2, D_2, E_2, \ldots$.

Notation. Dans la pratique, et quand le nombre des variables et étiquettes nécessaires est suffisamment petit, on omet les indices et on utilise les symboles X, Y, Z et A, B, C, D, E.

Définition 1.2 *Une* **action de** $\mathcal{S}$ *est une suite de symboles de l'une des trois formes suivantes :*

1. $V \leftarrow V + 1$ *(Action d'incrémentation)*
2. $V \leftarrow V - 1$ *(Action de décrémentation)*
3. *IF* $V \neq 0$ *GOTO L (Action de branchement conditionnel)*

où V *est une variable et* L *une étiquette.*

Remarque 1.1 Dans la définition 1.2 ci-dessus et jusqu'à la fin de ce chapitre, le symbole V désigne n'importe quelle variable de $\mathcal{S}$ mais n'est pas une variable du langage $\mathcal{S}$; c'est un symbole du métalangage dans lequel ces notes sont rédigées !

Définition 1.3 *Une* **instruction de** $\mathcal{S}$ *est soit une action*

1. $V \leftarrow V + 1$ *(Instruction d'incrémentation)*
2. $V \leftarrow V - 1$ *(Instruction de décrémentation)*
3. *IF* $V \neq 0$ *GOTO L (Instruction de branchement conditionnel)*

soit une action précédée d'une étiquette entre crochets :

4. $[L']\quad V \leftarrow V + 1$
5. $[L']\quad V \leftarrow V - 1$
6. $[L']$ *IF* $V \neq 0$ *GOTO L*

*On parle alors d'***instruction étiquetée** *et* L' *est l'***étiquette de l'instruction**.

Définition 1.4 *Un* **programme de** $\mathcal{S}$ *est une suite finie (éventuellement vide !) d'instructions (étiquetées ou non) qui satisfait la contrainte suivante : deux instructions étiquetées ne peuvent avoir la même étiquette*[1].

Le nombre d'instructions d'un programme $\mathcal{P}$ *est appelé la* **longueur de** $\mathcal{P}$.

Les variables qui apparaissent dans $\mathcal{P}$ *sont appelées les* **variables de** $\mathcal{P}$.

[1] Cette contrainte n'est pas indispensable. On ne l'impose dans ce chapitre que dans le but de clarifier l'écriture des programmes. Dans le chapitre 3, on la laissera tomber.

Exemple 1.1 (1) Voici un exemple de programme de longueur 3 :

$$
\begin{array}{ll}
[A] & X \leftarrow X - 1 \\
 & Y \leftarrow Y + 1 \\
 & \text{IF } X \neq 0 \text{ GOTO } A
\end{array}
$$

(2) Voici un exemple de programme de longueur 6, dans lequel on indique en outre le **numéro (d'ordre)** de chaque instruction dans le programme :

$$
\begin{array}{lll}
[A] & \text{IF } X \neq 0 \text{ GOTO } B & (1) \\
 & Z \leftarrow Z + 1 & (2) \\
 & \text{IF } Z \neq 0 \text{ GOTO } E & (3) \\
[B] & X \leftarrow X - 1 & (4) \\
 & Y \leftarrow Y + 1 & (5) \\
 & \text{IF } Y \neq 0 \text{ GOTO } A & (6)
\end{array}
$$

1.1.2 Sémantique du langage $\mathcal{S}$

Dans ce paragraphe, on précise la signification des programmes du langage $\mathcal{S}$.

Intuitivement, les variables sont des "boîtes" qui contiennent une valeur entière positive ou nulle, laquelle peut évoluer au cours de l'exécution d'un programme $\mathcal{P}$ de $\mathcal{S}$, qui se déroule de la façon suivante :

1. On donne aux variables d'entrée de $\mathcal{P}$ des valeurs d'entiers positifs ou nuls et on donne aux autres variables de $\mathcal{P}$ la valeur 0 ;

2. On "fait tourner" le programme sur ces valeurs initiales en commençant par la première instruction et en respectant l'interprétation des instructions donnée par le tableau ci-dessous. L'exécution se termine si on se trouve mis en demeure de poursuivre en exécutant une instruction qui n'existe pas.

Instruction	Interprétation
$V \leftarrow V + 1$	Ajouter 1 à la valeur de V et poursuivre en exécutant l'instruction suivante.
$V \leftarrow V - 1$	Si $V \neq 0$, ôter 1 à la valeur de V et poursuivre en exécutant l'instruction suivante ; Sinon, ne pas changer la valeur de V et poursuivre en exécutant l'instruction suivante.
IF $V \neq 0$ GOTO L	Si la valeur de V est non nulle, poursuivre en exécutant l'instruction d'étiquette L ; Sinon, poursuivre en exécutant l'instruction suivante.

Remarque 1.2 Les contraintes imposées (valeurs ≥ 0, interprétation de $V \leftarrow V - 1$ et de l'arrêt) sont de convention et permettent d'alléger la présentation de ce chapitre. On aurait pu faire le choix d'autres contraintes sans modifier les résultats fondamentaux du cours.

On va maintenant préciser les détails qui sont restés cachés dans l'interprétation "intuitive" des programmes et donner des définitions formelles.

Définition 1.5 *Soit $\mathcal{P}$ un programme de $\mathcal{S}$. Soit σ un ensemble d'équations de la forme $V = m$ où V est un symbole de variable et m un entier naturel.*

σ est un **état de** $\mathcal{P}$ *si et seulement si :*

1. *Pour chaque symbole de variable V apparaissant dans $\mathcal{P}$, σ contient une équation $V = m$.*
2. *Si $V = m$ et $V' = m'$ sont deux équations distinctes de σ, alors $V \neq V'$.*

Si σ contient l'équation $V = m$, on dit que m **est la valeur de V dans l'état σ de $\mathcal{P}$**[2].

Définition 1.6 *Soit $\mathcal{P}$ un programme de $\mathcal{S}$ de longueur n. Une* **description instantanée de** $\mathcal{P}$ *est un couple (i, σ) où i est un entier naturel inférieur ou égal à $n + 1$ et σ un état de $\mathcal{P}$. [Intuitivement, l'entier naturel i dans (i, σ) signifie que c'est la $i$ème instruction de $\mathcal{P}$ qui est sur le point d'être exécutée : $i = n + 1$ correspond à une "instruction d'arrêt" de l'exécution de $\mathcal{P}$.]*

Une description instantanée de $\mathcal{P}$ est dite **terminale** *si $i = n + 1$.*

---

[2]On aurait aussi pu définir un état comme une application (i.e. une fonction totale) de l'ensemble des variables dans $\mathbb{N}$ qui donne la valeur 0 à presque toutes les variables (i.e. à toutes les variables sauf un nombre fini.)

Définition 1.7 *Soit $\mathcal{P}$ un programme de $\mathcal{S}$ de longueur n et (i, σ) une description instantanée non terminale de $\mathcal{P}$ [i.e. $i \leq n$]. Le* **successeur** *de (i, σ) est la description instantanée (j, τ) définie de la façon suivante :*

Cas 1 : La ième instruction de $\mathcal{P}$ est "$V \leftarrow V + 1$" et σ contient l'équation $V = m$. Alors $j = i + 1$ et τ est obtenu en remplaçant $V = m$ par $V = m + 1$.

Cas 2 : La ième instruction de $\mathcal{P}$ est "$V \leftarrow V - 1$" et σ contient l'équation $V = m$. Alors $j = i + 1$ et, si $m \neq 0$, τ est obtenu en remplaçant $V = m$ par $V = m - 1$; si $m = 0$, $\tau = \sigma$.

Cas 3 : La ième instruction de $\mathcal{P}$ est "IF $V \neq 0$ GOTO L". Alors $\tau = \sigma$, et on distingue deux sous-cas :

Cas 3a : σ contient l'équation $V = 0$. Alors $j = i + 1$.

Cas 3b : σ contient l'équation $V = m$, où $m \neq 0$. Alors, s'il existe une instruction de $\mathcal{P}$ d'étiquette L, j est le numéro de cette instruction[3], sinon $j = n + 1$.

Définition 1.8 *Soit $\mathcal{P}$ un programme de $\mathcal{S}$. Un* **calcul de** $\mathcal{P}$ *est une suite finie $s_1, s_2, \ldots, s_k$ de descriptions instantanées de $\mathcal{P}$ telle que :*

1. *$s_1 = (1, \sigma)$, où σ est un état quelconque de $\mathcal{P}$ [i.e. on commence par exécuter la 1ère instruction de $\mathcal{P}$].*

2. *s_k est une description instantanée terminale.*

3. *Pour tout $i \in \{1, 2, \ldots, k - 1\}$, s_{i+1} est le successeur de s_i.*

Le nombre k s'appelle la **longueur du calcul** $s_1, s_2, \ldots, s_k$.

[3]Si on autorise les programmes dans lesquels plusieurs instructions portent la même étiquette (cf. la note de la définition 1.5), alors j est **le plus petit** $k \geq 1$ tel que l'instruction de numéro d'ordre k porte l'étiquette L.

Exemple 1.2 Voici un calcul du programme $\mathcal{P}$ de l'exemple 1.1(2) :

(1,{X=2,Y=0,Z=0})
(4,{X=2,Y=0,Z=0})
(5,{X=1,Y=0,Z=0})
(6,{X=1,Y=1,Z=0})
(1,{X=1,Y=1,Z=0})
(4,{X=1,Y=1,Z=0})
(5,{X=0,Y=1,Z=0})
(6,{X=0,Y=2,Z=0})
(1,{X=0,Y=2,Z=0})
(2,{X=0,Y=2,Z=0})
(3,{X=0,Y=2,Z=1})
(7,{X=0,Y=2,Z=1})

1.1.3 Les fonctions calculées par un programme

Définition 1.9 *Soit $\mathcal{P}$ un programme de $\mathcal{S}$ et $(r_1, r_2, \dots, r_m)$ un m-uplet d'entiers naturels. L'***état initial de $\mathcal{P}$ associé à** $(r_1, r_2, \dots, r_m)$ *est l'état de $\mathcal{P}$ constitué des équations*

$$X_1 = r_1, \;\; X_2 = r_2, \;\; \dots, \;\; X_m = r_m, \;\; Y = 0$$

et de l'équation $V = 0$ pour toute variable V de $\mathcal{P}$ n'appartenant pas à l'ensemble $\{X_1, X_2, \dots, X_m, Y\}$.

Définition 1.10 *Soit $\mathcal{P}$ un programme de $\mathcal{S}$ et m un entier naturel. La* **fonction de $\mathcal{F}_m$ calculée par** $\mathcal{P}$ *est la fonction $\psi_{\mathcal{P}}^{(m)}$ définie de la façon suivante :*

1er cas : Il y a un calcul $s_1, s_2, \dots, s_k$ de $\mathcal{P}$ tel que $s_1 = (1, \sigma)$, où σ est l'état initial de $\mathcal{P}$ associé au m-uplet $(r_1, r_2, \dots, r_m)$. Alors $\psi_{\mathcal{P}}^{(m)}(r_1, \dots, r_m)$ est la valeur de Y dans la description instantanée terminale s_k.

2ème cas : Un tel calcul n'existe pas : il existe une suite infinie[4] $s_1, s_2, \dots$ de descriptions instantanées de $\mathcal{P}$ telle que $s_1 = (1, \sigma)$, où σ est l'état initial de $\mathcal{P}$ associé au m-uplet $(r_1, r_2, \dots, r_m)$, et pour tout $i \geq 1$, s_{i+1} est le successeur de s_i [Aucun s_i n'est alors une description instantanée terminale]. Alors $\psi_{\mathcal{P}}^{(m)}(r_1, r_2, \dots, r_m)$ est indéfini et on écrit

[4] Par abus de langage, on parle parfois de calcul infini !

$$\psi_{\mathcal{P}}^{(m)}(r_1, r_2, \ldots, r_m) \uparrow .$$

Exemple 1.3 (1) La fonction de $\mathcal{F}_1$ calculée par le programme $\mathcal{P}$ de l'exemple 1.1(1) est :

$$\psi_{\mathcal{P}}^{(1)}(x) = \begin{cases} 1 & \text{si } x = 0 \\ x & \text{sinon.} \end{cases}$$

(2) La fonction de $\mathcal{F}_1$ calculée par le programme $\mathcal{P}$ de l'exemple 1.1(2) est :

$$\psi_{\mathcal{P}}^{(1)}(x) = x.$$

Définition 1.11 *Soit f une fonction (partielle) de $\mathcal{F}_m$. On dit que f est* **calculable** *s'il existe un programme $\mathcal{P}$ de $\mathcal{S}$ tel que $\psi_{\mathcal{P}}^{(m)} = f$ (on dit alors que $\mathcal{P}$ calcule f).*

Remarque 1.3 Par le vocabulaire choisi, on vient d'identifier assez brutalement la notion intuitive de "fonction calculable" et celle rigoureusement définie de "fonction calculable par un programme de $\mathcal{S}$". Les chapitres 2 et 3 achèveront de convaincre qu'on a été moins cavalier qu'il ne paraît !

1.2 L'utilisation des macros

Les programmes du langage $\mathcal{S}$ tel que nous l'avons défini dans les paragraphes précédents sont construits à partir d'un très petit ensemble d'actions (les incrémentations, les décrémentations et les branchements conditionnels). Cela peut rendre l'écriture d'un programme calculant une fonction donnée assez fastidieuse ; aussi allons nous autoriser l'emploi de **macro-instructions** (ou plus simplement, **macros**) qui sont des pseudo-instructions ayant le même effet qu'un bloc d'instructions de $\mathcal{S}$.

Une suite finie d'instructions de $\mathcal{S}$ et de macros constituera un **pseudo-programme**. Dans les sections 1.2.1 à 1.2.5, on définit les macros en précisant pour chacune d'elles le bloc d'instructions qu'elle abrège, appelé **expansion** de la macro. On pourra ainsi, en remplaçant dans un pseudo-programme $\hat{\mathcal{P}}$ chaque macro par son expansion, reconstruire un programme de $\mathcal{S}$, appelé **programme étendu** associé à $\hat{\mathcal{P}}$, qui calcule la même fonction que $\hat{\mathcal{P}}$.

On remarquera que la licéité de l'emploi de certaines macros dépend de la démonstration préalable de la calculabilité d'une fonction ou d'un prédicat (cf. les sections 1.2.4 et 1.2.5).

1.2.1 Le branchement inconditionnel

Soit $\mathcal{P}$ un programme de $\mathcal{S}$, de longueur n. Supposons que pour un $i \in \{1, \ldots, n-1\}$ les $i$ème et $(i+1)$ème instructions de $\mathcal{P}$ soient

$$\begin{array}{ll} Z \leftarrow Z+1 & (i) \\ \text{IF } Z \neq 0 \text{ GOTO } L & (i+1) \end{array}$$

et que la variable auxiliaire Z n'apparaisse dans aucune autre instruction de $\mathcal{P}$.

Lors d'un calcul de $\mathcal{P}$, si on doit exécuter la $i$ème instruction, alors on ajoutera 1 à la valeur z de Z et la condition $Z \neq 0$ de la $(i+1)$ème instruction sera toujours satisfaite. L'effet de cette suite de deux instructions est donc le même, à la modification de z près, que celui de la pseudo-instruction

$$\text{GOTO } L$$

dont l'interprétation naturelle est "poursuivre par l'instruction d'étiquette L". Comme Z n'apparaît dans $\mathcal{P}$ que dans les $i$ème et $(i+1)$ème instructions, la fonction de $\mathcal{F}_m$ calculée par $\mathcal{P}$ ne sera nullement affectée par l'abréviation de

$$\begin{array}{l} Z \leftarrow Z+1 \\ \text{IF } Z \neq 0 \text{ GOTO } L \end{array}$$

en

$$\text{GOTO } L$$

La pseudo-instruction "GOTO L" est une **macro** (appelée **branchement inconditionnel**) dont l'**expansion** est

$$\begin{array}{l} Z \leftarrow Z+1 \\ \text{IF } Z \neq 0 \text{ GOTO } L \end{array}$$

On s'autorisera dorénavant à utiliser cette macro pour écrire des pseudo-programmes de $\mathcal{S}$ comme s'il s'agissait d'une instruction du langage $\mathcal{S}$ en sachant que pour recouvrer un véritable programme de $\mathcal{S}$, il suffirait de remplacer chaque occurrence de

$$\text{GOTO } L$$

par

$$\begin{array}{l} Z \leftarrow Z+1 \\ \text{IF } Z \neq 0 \text{ GOTO } L \end{array}$$

où Z est une variable non utilisée dans le pseudo-programme.

Exemple 1.4 Voici un pseudo-programme calculant la fonction identité qui abrège (un peu) le programme de l'exemple 1.1(2) :

$$\begin{array}{ll} [A] & \text{IF } X \neq 0 \text{ GOTO } B \\ & \text{GOTO } E \\ [B] & X \leftarrow X - 1 \\ & Y \leftarrow Y + 1 \\ & \text{GOTO } A \end{array}$$

[On remarque que l'expansion du pseudo-programme ci-dessus ne donne pas exactement le programme de l'exemple 1.1(2).]

1.2.2 La mise à zéro

Soit un programme $\mathcal{P}$ qui contient les deux instructions consécutives suivantes :

$$\begin{array}{ll} [L] & V \leftarrow V - 1 \\ & \text{IF } V \neq 0 \text{ GOTO } L \end{array}$$

Lors d'un calcul de $\mathcal{P}$, l'effet de cette suite de deux instructions est de donner la valeur 0 à la variable V. On introduit, pour toute variable V, une macro (appelée **mise à zéro**)

$$V \leftarrow 0$$

dont l'interprétation est "donner à V la valeur 0" et dont l'expansion est

$$\begin{array}{ll} [L] & V \leftarrow V - 1 \\ & \text{IF } V \neq 0 \text{ GOTO } L \end{array}$$

Pour cette nouvelle série de macros, la seule précaution à prendre lorsqu'on leur substitue leur expansion dans un pseudo-programme est de choisir une étiquette L non utilisée dans le pseudo-programme si la macro n'est pas étiquetée.

1.2.3 L'affectation

Soit maintenant un pseudo-programme $\hat{\mathcal{P}}$, de longueur $n \geq 12$, qui contient la suite de 12 instructions ou macros suivantes $(0 \leq i < n - 11)$:

$$
\begin{array}{lll}
 & V \leftarrow 0 & (i{+}1) \\
[A] & \text{IF } V' \neq 0 \text{ GOTO } B & (i{+}2) \\
 & \text{GOTO } C & (i{+}3) \\
[B] & V' \leftarrow V' - 1 & (i{+}4) \\
 & V \leftarrow V + 1 & (i{+}5) \\
 & Z \leftarrow Z + 1 & (i{+}6) \\
 & \text{GOTO } A & (i{+}7) \\
[C] & \text{IF } Z \neq 0 \text{ GOTO } D & (i{+}8) \\
 & \text{GOTO } E & (i{+}9) \\
[D] & Z \leftarrow Z - 1 & (i{+}10) \\
 & V' \leftarrow V' + 1 & (i{+}11) \\
 & \text{GOTO } C & (i{+}12)
\end{array}
$$

où Z est une variable auxiliaire qui n'apparaît dans aucune autre instruction (ou pseudo-instruction) de $\hat{\mathcal{P}}$ et E est l'étiquette de la $(i{+}13)$ème instruction (ou pseudo-instruction) de $\hat{\mathcal{P}}$, si elle existe.

Lors d'un calcul de $\hat{\mathcal{P}}$, l'effet de ce bloc de (pseudo-)instructions est de donner à V la valeur de V' sans modifier cette dernière. Pour tout couple de variables V, V', on introduit une macro

$$V \leftarrow V'$$

dont l'interprétation est "donner à V la valeur de V' et poursuivre par l'instruction qui suit immédiatement le bloc" et dont l'expansion est précisément le bloc de 12 instructions ci-dessus.

Quand on substitue dans un pseudo-programme $\hat{\mathcal{P}}$ son expansion à la macro $V \leftarrow V'$, il faut veiller à respecter 3 contraintes :

1. Choisir une variable Z non utilisée dans $\hat{\mathcal{P}}$;
2. Choisir des étiquettes A, B, C et D non utilisées dans $\hat{\mathcal{P}}$;
3. Si l'instruction qui suit immédiatement la macro $V \leftarrow V'$ dans $\hat{\mathcal{P}}$ existe et porte une étiquette, choisir cette étiquette en lieu et place de E à la 9ème ligne de l'expansion ; sinon choisir une étiquette E qui n'apparaît pas dans $\hat{\mathcal{P}}$ et étiqueter l'instruction qui suit immédiatement la macro, si elle existe, par E.

Exemple 1.5 (1) Voici un pseudo-programme calculant la fonction Add $\in \mathcal{F}_2$ définie par $\mathrm{Add}(x_1, x_2) = x_1 + x_2$:

$$\begin{array}{ll} & Y \leftarrow X_1 \\ [B] & \text{IF } X_2 \neq 0 \text{ GOTO } A \\ & \text{GOTO } E \\ [A] & X_2 \leftarrow X_2 - 1 \\ & Y \leftarrow Y + 1 \\ & \text{GOTO } B \end{array}$$

(2) On remarque que l'exécution du pseudo-programme ci-dessus ne préserve pas la valeur initiale d'une des deux variables d'entrée (X_2). Pour remédier à cet "inconvénient", on peut choisir le pseudo-programme suivant, qui calcule exactement la même fonction de $\mathcal{F}_2$:

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

1.2.4 La macro $W \leftarrow f(V_1, \ldots, V_n)$

Soit f une fonction (partielle) de $\mathcal{F}_n$. Si f est calculable, il existe un programme de $\mathcal{S}$ qui la calcule. Il est facile de renommer les variables de ce programme de façon que les contraintes suivantes soient satisfaites :

1. Toutes les variables du programmes appartiennent à l'ensemble $\{Y, X_1, \ldots, X_n, Z_1, \ldots, Z_k\}$;
2. Toutes les étiquettes utilisées dans le programme appartiennent à l'ensemble $\{E, A_1, \ldots, A_\ell\}$;
3. E est la seule "étiquette de sortie", autrement dit : si "IF $V \neq 0$ GOTO A_i" ($1 \leq i \leq \ell$) est une instruction du programme, alors il existe dans le programme une instruction étiquetée par A_i.

Soit $\mathcal{Q}$ un programme, de longueur q, calculant f et satisfaisant les 3 contraintes ci-dessus. Soit maintenant un pseudo-programme $\hat{\mathcal{P}}$, de longueur

$\geq n + k + q + 2$ contenant les lignes suivantes :

	$Z_m \leftarrow 0$	ligne $(i+1)$
	$Z_{m+1} \leftarrow V_1$	ligne $(i+2)$
	$Z_{m+2} \leftarrow V_2$	ligne $(i+3)$
	$\vdots$	lignes $(i+4)$ à $(i+n)$
	$Z_{m+n} \leftarrow V_n$	ligne $(i+n+1)$
	$Z_{m+n+1} \leftarrow 0$	ligne $(i+n+2)$
	$Z_{m+n+2} \leftarrow 0$	ligne $(i+n+3)$
	$\vdots$	lignes $(i+n+4)$ à $(i+k)$
	$Z_{m+n+k} \leftarrow 0$	ligne $(i+n+k+1)$
	$\mathcal{Q}_m$	lignes $(i+n+k+2)$ à $(i+n+k+q+1)$
$[E_m]$	$W \leftarrow Z_m$	ligne $(i+n+k+q+2)$

où $\mathcal{Q}_m$ est le programme obtenu en remplaçant **simultanément** dans $\mathcal{Q}$

- toutes les occurrences de Y par Z_m,
- toutes les occurrences de X_i par Z_{m+i}, pour $i \in \{1, \ldots, n\}$,
- toutes les occurrences de Z_j par Z_{m+n+j}, pour $j \in \{1, \ldots, k\}$,
- toutes les occurrences de E par E_m et
- toutes les occurrences de A_p par A_{m+p}, pour $p \in \{1, \ldots, \ell\}$.

et où m est choisi suffisamment grand pour qu'aucune variable ou étiquette de $\mathcal{Q}_m$ n'apparaisse dans le pseudo-programme $\hat{\mathcal{P}}$ en dehors des lignes $(i+1)$ à $(i+n+k+q+2)$ [En particulier, les ensembles $\{W, V_1, \ldots, V_n\}$ et $\{Z_m, \ldots, Z_{m+n+k}\}$ sont disjoints.]

Si, au cours de l'exécution du pseudo-programme $\hat{\mathcal{P}}$, on doit exécuter la première (pseudo-)instruction du bloc d'instructions et de pseudo-instructions ci-dessus, ce dernier aura l'effet suivant :

1. Donner aux variables $Z_m, Z_{m+n+1}, \ldots, Z_{m+n+k}$ la valeur 0 ; donner aux variables $Z_{m+1}, \ldots, Z_{m+n}$ les valeurs respectives $v_1, \ldots, v_n$ de $V_1, \ldots, V_n$ sans modifier ces dernières ;

2. "calculer" $f(v_1, \ldots, v_n)$ à l'aide du programme $\mathcal{Q}_m$
   1er cas : $f(v_1, \ldots, v_n)$ est défini. Alors on donne à la variable W la valeur $f(v_1, \ldots, v_n)$ et on poursuit par l'instruction $(i+n+k+q+3)$ de $\hat{\mathcal{P}}$ si cette instruction existe, on s'arrête sinon ;
   2ème cas : $f(v_1, \ldots, v_n)$ n'est pas défini. Alors l'exécution de $\hat{\mathcal{P}}$ ne s'arrête pas (par définition de $\mathcal{Q}_m$, car $\mathcal{Q}$ calcule f !)

Pour tout $n \geq 1$ et toute fonction partielle calculable f de $\mathcal{F}_n$, on introduit une nouvelle macro, qu'on note

$$W \leftarrow f(V_1, \ldots, V_n)$$

dont l'interprétation est

> "Si $v_1, \ldots, v_n$ désignent respectivement les valeurs courantes des variables $V_1, \ldots, V_n$ et si $f(v_1, \ldots, v_n)$ est défini, alors donner à la variable W la valeur $f(v_1, \ldots, v_n)$ et poursuivre par l'instruction suivant immédiatement la macro.
>
> Si $v_1, \ldots, v_n$ désignent respectivement les valeurs courantes des variables $V_1, \ldots, V_n$ et si $f(v_1, \ldots, v_n) \uparrow$, alors poursuivre indéfiniment l'exécution de la macro..."

et dont l'expansion est (l'expansion de)

$$\begin{array}{ll}
 & Z_m \leftarrow 0 \\
 & Z_{m+1} \leftarrow V_1 \\
 & Z_{m+2} \leftarrow V_2 \\
 & \vdots \\
 & Z_{m+n} \leftarrow V_n \\
 & Z_{m+n+1} \leftarrow 0 \\
 & Z_{m+n+2} \leftarrow 0 \\
 & \vdots \\
 & Z_{m+n+k} \leftarrow 0 \\
 & \mathcal{Q}_m \\
[E_m] & W \leftarrow Z_m
\end{array}$$

Lors du remplacement de la macro "$W \leftarrow f(V_1, \ldots, V_n)$" par son expansion, il faut veiller à respecter les contraintes pesant sur m et le programme $\mathcal{Q}_m$.

Remarque 1.4 Il peut sembler inutile de donner aux variables $Z_m, Z_{m+n+1}, \ldots,\ Z_{m+n+k}$ la valeur 0 au début de l'expansion, puisqu'il s'agit de variables auxiliaires n'apparaissant que dans l'expansion de la macro. En vérité, cette précaution est indispensable si l'exécution du pseudo-programme $\hat{\mathcal{P}}$ contenant la macro impose plusieurs exécutions de la macro ; en effet, les valeurs de $Z_m, Z_{m+n+1}, \ldots, Z_{m+n+k}$ peuvent être modifiées par l'exécution de $\mathcal{Q}_m$!

Il est tout aussi indispensable de "transvaser" les variables d'entrée de $\hat{\mathcal{Q}}$ dans $Z_m, \ldots, Z_{m+n}$, de travailler sur ces dernières, puis de "transvaser" le résultat dans Y (cf. l'exemple 1.6 ci-dessous.)

Exemple 1.6 Soit Add la fonction de $\mathcal{F}_2$ définie par $\mathrm{Add}(x_1, x_2) = x_1 + x_2$. Cette fonction est calculable (cf. l'exemple 1.5.) Le pseudo-programme $\hat{\mathcal{P}}$ suivant calcule la fonction Mult $\in \mathcal{F}_2$ définie par $\mathrm{Mult}(x_1, x_2) = x_1 \cdot x_2$:

$$
\begin{array}{lll}
 & Z \leftarrow X_2 & (1) \\
[B] & \text{IF } Z \neq 0 \text{ GOTO } A & (2) \\
 & \text{GOTO } E & (3) \\
[A] & Z \leftarrow Z - 1 & (4) \\
 & Y \leftarrow \mathrm{Add}(X_1, Y) & (5) \\
 & \text{GOTO } B & (6)
\end{array}
$$

On cherche maintenent à remplacer "$Y \leftarrow \mathrm{Add}(X_1, Y)$" dans $\hat{\mathcal{P}}$ par son expansion en utilisant le pseudo-programme de l'exemple 1.5(2). Après renommage des étiquettes, on obtient un "nouveau" pseudo-programme $\hat{\mathcal{Q}}$ calculant l'addition :

$$
\begin{array}{ll}
 & Y \leftarrow X_1 \\
 & Z \leftarrow X_2 \\
[A_2] & \text{IF } Z \neq 0 \text{ GOTO } A_1 \\
 & \text{GOTO } E \\
[A_1] & Z \leftarrow Z - 1 \\
 & Y \leftarrow Y + 1 \\
 & \text{GOTO } A_2
\end{array}
$$

On peut prendre $m = 2$ et remplacer la ligne (5) de $\hat{\mathcal{P}}$ par son expansion pour obtenir

$$
\begin{array}{lll}
 & Z_1 \leftarrow X_2 & (1) \\
[B_1] & \text{IF } Z_1 \neq 0 \text{ GOTO } A_1 & (2) \\
 & \text{GOTO } E_1 & (3) \\
[A_1] & Z_1 \leftarrow Z_1 - 1 & (4) \\
 & Z_2 \leftarrow 0 & (5) \\
 & Z_3 \leftarrow X_1 & (6) \\
 & Z_4 \leftarrow Y & (7) \\
 & Z_5 \leftarrow 0 & (8) \\
 & Z_2 \leftarrow Z_3 & (9) \\
 & Z_5 \leftarrow Z_4 & (10) \\
[A_3] & \text{IF } Z_5 \neq 0 \text{ GOTO } A_2 & (11) \\
 & \text{GOTO } E_2 & (12) \\
[A_2] & Z_5 \leftarrow Z_5 - 1 & (13) \\
 & Z_2 \leftarrow Z_2 + 1 & (14) \\
 & \text{GOTO } A_3 & (15) \\
[E_2] & Y \leftarrow Z_2 & (16) \\
 & \text{GOTO } B_1 & (17)
\end{array}
$$

[L'expansion correspond aux lignes (5) à (16).] On remarque d'abord que la ligne (16) est indispensable car la valeur de Y n'est pas modifiée par l'exécution des lignes (5) à (15). D'autre part, dans ce cas précis, les lignes (5) à (10) peuvent être abrégées par les deux lignes suivantes :

$$
\begin{array}{l}
Z_2 \leftarrow X_1 \\
Z_5 \leftarrow Y
\end{array}
$$

Les variables Z_3 et Z_4 (qui renomment les variables X_1 et X_2 de $\hat{\mathcal{Q}}$) sont donc inutiles dans l'expansion. Toutefois, cela ne doit pas laisser supposer qu'il est suffisant, dans le remplacement de la macro

$$V' \leftarrow f(V_1, \ldots, V_n)$$

par son expansion, d'éviter les interférences entre variables auxiliaires et entre étiquettes, puis de remplacer les variables d'entrée $X_1, \ldots, X_n$ du (pseudo-)programme calculant f par les variables $V_1, \ldots, V_n$. En effet, ce procédé d'insertion de $\hat{\mathcal{Q}}$ dans $\hat{\mathcal{P}}$ (après renommage de Z en Z_2) conduirait au programme suivant

$$
\begin{array}{lll}
 & Z_1 \leftarrow X_2 & (1) \\
[B_1] & \text{IF } Z_1 \neq 0 \text{ GOTO } A_1 & (2) \\
 & \text{GOTO } E_1 & (3) \\
[A_1] & Z_1 \leftarrow Z_1 - 1 & (4) \\
 & Y \leftarrow X_1 & (5) \\
 & Z_2 \leftarrow Y & (6) \\
[A_3] & \text{IF } Z_2 \neq 0 \text{ GOTO } A_2 & (7) \\
 & \text{GOTO } E_2 & (8) \\
[A_2] & Z_2 \leftarrow Z_2 - 1 & (9) \\
 & Y \leftarrow Y + 1 & (10) \\
 & \text{GOTO } A_3 & (11) \\
[E_2] & \text{GOTO } B_1 & (12)
\end{array}
$$

qui calcule la fonction $(x_1, x_2) \mapsto 2x_1$!

1.2.5 La macro IF $P(V_1, \ldots, V_n)$ GOTO L

Définition 1.12 *Soit $n \geq 1$. Un* **prédicat d'arité** *n (ou* **prédicat n-aire***) est un sous-ensemble de $\mathbb{N}^n$. Soit P un prédicat n-aire et $(x_1, x_2, \ldots, x_n)$ un n-uplet d'entiers naturels. On écrit $P(x_1, x_2, \ldots, x_n)$ pour signifier $(x_1, x_2, \ldots, x_n) \in P$ et $\neg P(x_1, x_2, \ldots, x_n)$ pour signifier $(x_1, x_2, \ldots, x_n) \notin P$. On dit que P* **est calculable** *si la fonction (totale) $\mathbf{1}_P$ de $\mathcal{F}_n$ définie par*

$$\mathbf{1}_P(x_1, \ldots, x_n) = \begin{cases} 1 & si\ P(x_1, x_2, \ldots, x_n) \\ 0 & sinon \end{cases}$$

est calculable. La fonction $\mathbf{1}_P$ est appelée la **fonction indicatrice** *de P.*

Soit maintenant P un prédicat n-aire calculable et un pseudo-programme $\hat{\mathcal{P}}$ contenant le bloc de deux (pseudo-)instructions suivant :

$$
\begin{array}{l}
Z \leftarrow \mathbf{1}_P(V_1, \ldots, V_n) \\
\text{IF } Z \neq 0 \text{ GOTO } L
\end{array}
$$

où Z est une variable qui n'apparaît que dans les deux (pseudo-)instructions ci-dessus.

Si, au cours de l'exécution du pseudo-programme $\hat{\mathcal{P}}$, on doit exécuter la première instruction du bloc d'instructions ci-dessus, ce dernier aura l'effet suivant :

1. Donner à la variable Z la valeur $\mathbf{1}_P(v_1, \ldots, v_n)$ où $v_1, \ldots, v_n$ désignent respectivement les valeurs courantes des variables $V_1, \ldots, V_n$ ($\mathbf{1}_P(v_1, \ldots, v_n)$ est toujours défini car $\mathbf{1}_P$ est une fonction totale!) ;
2. 1er cas : $\mathbf{1}_P(v_1, \ldots, v_n) = 1$. Alors on poursuit par l'instruction d'étiquette L si cette instruction existe, on s'arrête sinon ;
   2ème cas : $\mathbf{1}_P(v_1, \ldots, v_n) = 0$. Alors on poursuit par l'instruction qui suit immédiatement le bloc si cette instruction existe, on s'arrête sinon.

Pour tout $n \geq 1$ et tout prédicat n-aire calculable P, on introduit une nouvelle macro, qu'on note

$$\text{IF } P(V_1, \ldots, V_n) \text{ GOTO } L$$

dont l'interprétation est

> "Si $v_1, \ldots, v_n$ désignent respectivement les valeurs courantes des variables $V_1, \ldots, V_n$ et si $\neg P(v_1, \ldots, v_n)$, alors poursuivre par l'instruction suivant immédiatement la macro.
>
> Si $v_1, \ldots, v_n$ désignent respectivement les valeurs courantes des variables $V_1, \ldots, V_n$ et si $P(v_1, \ldots, v_n)$, alors poursuivre par l'instruction d'étiquette L"

et dont l'expansion est (l'expansion de)

$$\begin{array}{l} Z \leftarrow \mathbf{1}_P(V_1, \ldots, V_n) \\ \text{IF } Z \neq 0 \text{ GOTO } L \end{array}$$

Exemple 1.7 (1) Le prédicat binaire P défini par

$$P(x_1, x_2) \overset{déf}{\iff} x_1 = x_2$$

est calculé par

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

Le prédicat unaire P défini par

$$P(x) \stackrel{déf}{\Longleftrightarrow} x = 0$$

est calculé par

$$\begin{array}{l} \text{IF } X \neq 0 \text{ GOTO } E \\ Y \leftarrow Y + 1 \end{array}$$

On peut donc utiliser les macros

$$\text{IF } V = V' \text{ GOTO } L \qquad \text{et} \qquad \text{If } V = 0 \text{ GOTO } L$$

pour écrire le pseudo-programme $\hat{\mathcal{P}}$ suivant :

$$\begin{array}{ll} & \text{IF } X = 0 \text{ GOTO } A \\ & Z_1 \leftarrow X \\ & Z_1 \leftarrow Z_1 - 1 \\ & \text{IF } Z_1 = 0 \text{ GOTO } A \\ [B] & Z_1 \leftarrow Z_1 - 1 \\ & \text{IF } Z_1 = 0 \text{ GOTO } E \\ & Z_2 \leftarrow \text{Mult}(Z_1, Z_1) \\ & \text{IF } Z_2 = X \text{ GOTO } A \\ & \text{GOTO } B \\ [A] & Y \leftarrow Y + 1 \end{array}$$

où Mult dénote la multiplication (cf. l'exemple 1.6 ci-dessus). On vérifie aisément que $\hat{\mathcal{P}}$ calcule le prédicat P défini par

$$P(x) \stackrel{déf}{\Longleftrightarrow} x \text{ est un carré parfait}$$

Chapitre 2

Fonctions (primitives) récursives

Dans ce chapitre, on définit et étudie deux ensembles de fonctions (partielles). Le premier, l'ensemble des fonctions primitives récursives, est constitué de fonctions totales calculables ; à vrai dire, il contient toutes les fonctions (nécessairement totales) qu'on sait "effectivement calculer sur machine" à ce jour ! Le second ensemble, celui des fonctions (partielles) récursives, est égal à l'ensemble des fonctions calculables défini dans le chapitre 1.

2.1 Les fonctions primitives récursives

2.1.1 Le schéma de composition

Définition 2.1 (Schéma de composition) *Soit f une fonction de $\mathcal{F}_k$ et $g_1, \ldots, g_k$ des fonctions de $\mathcal{F}_n$. La fonction obtenue par* **composition de $g_1, \ldots, g_k$ par** *f est la fonction h de $\mathcal{F}_n$ définie par*

$$h(x_1, \ldots, x_n) = f(g_1(x_1, \ldots, x_n), \ldots, g_k(x_1, \ldots, x_n)).$$

Notation. La fonction h de la définition 2.1 est appelée **composée de $g_1, \ldots, g_k$ par** f. On utilise la notation $f[g_1, \ldots, g_k]$ (ou $f(g_1, \ldots, g_k)$) pour la désigner. Dans le cas où f est une fonction unaire, la composée de g par f est la fonction usuellement notée $f \circ g$.

Remarque 2.1 La définition 2.1 peut se réécrire $h = f[g_1, \ldots, g_k]$. Comme d'habitude, l'égalité entre deux fonctions signifie : pour un argument donné (en l'occurrence, un n-uplet d'entiers naturels), ou bien aucune des deux fonctions n'est définie, ou bien les deux fonctions sont définies et prennent la même valeur.

Proposition 2.1 *Si la fonction $f \in \mathcal{F}_k$ et toutes les fonctions $g_1, \ldots, g_k \in \mathcal{F}_n$ sont totales, alors la fonction $h = f[g_1, \ldots, g_k] \in \mathcal{F}_n$ est totale.* ■

Remarque 2.2 Si l'une des fonctions $g_1, \ldots, g_k$ n'est pas totale, alors h n'est pas totale. En revanche, h peut être totale alors que f ne l'est pas ; c'est le cas quand les valeurs prises par les fonctions $g_1, \ldots, g_k$ sont telles que pour tous entiers naturels $x_1, \ldots, x_n$, le k-uplet $(g_1(x_1, \ldots, x_n), \ldots, g_k(x_1, \ldots, x_n))$ appartient à l'ensemble de définition de f (cf. l'exemple 2.2 ci-dessous).

Autrement dit, $h(x_1, \ldots, x_n)$ est défini si et seulement si $g_1(x_1, \ldots, x_n), \ldots,$ $g_k(x_1, \ldots, x_n)$ sont tous définis et $f(z_1, \ldots, z_k)$, où $z_1 = g_1(x_1, \ldots, x_n), \ldots, z_k = g_k(x_1, \ldots, x_n)$, est défini.

Exemple 2.1 La fonction h de $\mathcal{F}_3$ définie par

$$h(x_1, x_2, x_3) = \begin{cases} (x_1 - x_2) + x_3 & \text{si } x_1 \geq x_2 \\ \uparrow & \text{si } x_1 < x_2 \end{cases}$$

est la composée de g_1, u_3^3 par Add où g_1 et u_3^3 sont les fonctions de $\mathcal{F}_3$ définies par

$$g_1(x_1, x_2, x_3) = \begin{cases} (x_1 - x_2) & \text{si } x_1 \geq x_2 \\ \uparrow & \text{si } x_1 < x_2 \end{cases} \qquad \text{et} \qquad u_3^3(x_1, x_2, x_3) = x_3$$

et Add est la fonction de $\mathcal{F}_2$ définie par

$$\mathrm{Add}(x_1, x_2) = x_1 + x_2.$$

Exemple 2.2 La fonction unaire h définie par

$$h(x) = 4x^2 - 2x$$

est la composée de g_1, g_2 par Moins où g_1 et g_2 sont les fonctions unaires définies par

$$g_1(x) = 4x^2 \qquad \text{et} \qquad g_2(x) = 2x$$

et Moins est la fonction binaire définie par

$$\mathrm{Moins}(x_1, x_2) = \begin{cases} (x_1 - x_2) & \text{si } x_1 \geq x_2 \\ \uparrow & \text{si } x_1 < x_2 \end{cases}$$

Exemple 2.3 La fonction $h \in \mathcal{F}_3$ définie par

$$h(x_1, x_2, x_3) = x_3 + x_1$$

est la composée de u_3^3, u_1^3 par l'addition Add, où u_3^3 et u_1^3 désignent les fonctions de $\mathcal{F}_3$ définies par

$$u_3^3(x_1, x_2, x_3) = x_3 \qquad \text{et} \qquad u_1^3(x_1, x_2, x_3) = x_1.$$

Théorème 2.2 *Soit f une fonction de $\mathcal{F}_k$ et $g_1, \ldots, g_k$ des fonctions de $\mathcal{F}_n$. Si $g_1, \ldots, g_k$ et f sont toutes calculables, alors la composée h de $g_1, \ldots, g_k$ par f est une fonction calculable de $\mathcal{F}_n$.*

Démonstration : Les fonctions $g_1, \ldots, g_k$ et f étant calculables, on peut librement utiliser les macros d'affectation fonctionnelle qui leur sont associées pour écrire le pseudo-programme suivant :

$$\begin{array}{l} Z_1 \leftarrow g_1(X_1, \ldots, X_n) \\ \vdots \\ Z_k \leftarrow g_k(X_1, \ldots, X_n) \\ Y \leftarrow f(Z_1, \ldots, Z_k) \end{array}$$

Il est facile de vérifier que ce programme calcule h. ■

2.1.2 Le schéma de récursion primitive

Définition 2.2 (Schéma de récursion primitive[1]) *Soit $n \geq 1$, f une fonction de $\mathcal{F}_n$ et g une fonction de $\mathcal{F}_{n+2}$. La fonction obtenue par* **récursion primitive à partir de** *f* **et** *g est la fonction h de $\mathcal{F}_{n+1}$ définie par*

$$\left\{ \begin{array}{lcl} h(x_1, \ldots, x_n, 0) & = & f(x_1, \ldots, x_n) \\ h(x_1, \ldots, x_n, t+1) & = & g(x_1, \ldots, x_n, t, h(x_1, \ldots, x_n, t)). \end{array} \right.$$

Par convention, une fonction 0-aire est constante et la fonction obtenue par **récursion primitive à partir de** *m (où m est un entier naturel fixé)* **et** *$g \in \mathcal{F}_2$ est la fonction h de $\mathcal{F}_1$ définie par*

$$\left\{ \begin{array}{lcl} h(0) & = & m \\ h(t+1) & = & g(t, h(t)). \end{array} \right.$$

[1] On peut également parler de **schéma de récurrence**.

Notation. On utilise parfois la notation $\mathrm{rec}(f,g)$ pour désigner la fonction obtenue par récursion primitive à partir de f et g. Dans le cas où g est une fonction de $\mathcal{F}_2$, on écrit $\mathrm{rec}(m,g)$ pour désigner la fonction de $\mathcal{F}_1$ obtenue par récursion primitive à partir de m et g.

Proposition 2.3 *Si les fonctions $f \in \mathcal{F}_n$ et $g \in \mathcal{F}_{n+2}$ sont totales, alors la fonction $h = rec(f,g) \in \mathcal{F}_{n+1}$ est totale.*

Démonstration : Soit $x_1, \dots, x_n$ un n-uplet d'entiers naturels quelconque.

(a) $h(x_1, \dots, x_n, 0)$ est défini car f est totale.

(b) Soit t un entier naturel quelconque. Supposons que $h(x_1, \dots, x_n, t)$ soit défini. Comme la fonction g est totale, $g(x_1, \dots, x_n, t, h(x_1, \dots, x_n, t))$ est défini et $h(x_1, \dots, x_n, t+1)$ également !

On montre ainsi (par **récurrence sur** t) que pour tout n-uplet d'entiers naturels $(x_1, \dots, x_n)$ et tout entier naturel t, $h(x_1, \dots, x_n, t)$ est défini. La fonction h est donc totale. ■

Remarque 2.3 S'il existe un n-uplet $(x_1, \dots, x_n)$ tel que $f(x_1, \dots, x_n)$ n'est pas défini, alors $h(x_1, \dots, x_n, x_{n+1})$ n'est défini pour aucun entier naturel x_{n+1}. Si $h(x_1, \dots, x_n, x_{n+1})$ n'est pas défini, alors $h(x_1, \dots, x_n, y)$ n'est défini pour aucun entier naturel $y \geq x_{n+1}$. En revanche, h peut être totale sans que g le soit.

Exemple 2.4 La multiplication $\mathrm{Mult} \in \mathcal{F}_2$ définie par $\mathrm{Mult}(x_1, x_2) = x_1 \cdot x_2$ est obtenue par récursion primitive à partir de 0 et h où $h = \mathrm{Add}[u_3^3, u_1^3]$ est la fonction de l'exemple 2.3. En effet,

$$\begin{cases} x \cdot 0 & = & 0 \\ x \cdot (t+1) & = & (x \cdot t) + x & = & \mathrm{Add}(u_3^3(x,t,x \cdot t), u_1^3(x,t,x \cdot t)). \end{cases}$$

Théorème 2.4 *Soit f une fonction de $\mathcal{F}_n$ et g une fonction de $\mathcal{F}_{n+2}$. Si f et g sont calculables, alors la fonction h obtenue par récursion primitive à partir de f et g est calculable.*

Démonstration : Soit le pseudo-programme $\mathcal{P}$ suivant :

$$\begin{array}{lll} & Y \leftarrow f(X_1, \dots, X_n) & (1) \\ [A] & \text{IF } X_{n+1} = 0 \text{ GOTO } E & (2) \\ & Y \leftarrow g(X_1, \dots, X_n, Z, Y) & (3) \\ & Z \leftarrow Z + 1 & (4) \\ & X_{n+1} \leftarrow X_{n+1} - 1 & (5) \\ & \text{GOTO } A & (6) \end{array}$$

Par convention, $x_1, \ldots, x_n, x_{n+1}$ désignent respectivement les valeurs initiales des variables d'entrée $X_1, \ldots, X_n, X_{n+1}$. On va montrer que pour tous $x_1, \ldots, x_n$, l'exécution de $\mathcal{P}$ s'arrête avec une valeur finale de Y égale à $h(x_1, \ldots, x_n, x_{n+1})$ si $h(x_1, \ldots, x_n, x_{n+1})$ est défini, et ne s'arrête pas sinon. [Dans la suite de cette démonstration, on abrègera $x_1, \ldots, x_n$ en $\overline{x}$] :

(a) Si $x_{n+1} = 0$, alors l'exécution de $\mathcal{P}$ s'arrête si et seulement si $f(\overline{x})$ est défini et la valeur finale de Y est $f(\overline{x})$, c'est-à-dire $h(x_1, \ldots, x_n, x_{n+1})$.

(b) Si $x_{n+1} \geq 1$. On distingue deux cas.

1er cas : $h(\overline{x}, x_{n+1})$ est défini. Alors $h(\overline{x}, 0)$ –c'est-à-dire $f(\overline{x})$– est défini et pour tout entier naturel $1 \leq z \leq x_{n+1}$, $h(\overline{x}, z)$ –c'est-à-dire $g(\overline{x}, z-1, h(\overline{x}, z-1))$– est défini (cf. la remarque 2.3). L'exécution de $\mathcal{P}$ commence donc par affecter à Y la valeur de $f(\overline{x})$. En observant que si la valeur courante de X_{n+1} est non nulle et si Y a la valeur $h(\overline{x}, z)$ avant l'exécution de l'instruction d'étiquette A, alors elle a la valeur $g(\overline{x}, z, h(\overline{x}, z))$ –c'est-à-dire $h(\overline{x}, z+1)$– après l'exécution de l'instruction de numéro (3), on se convainc que l'exécution de $\mathcal{P}$ se poursuit en donnant successivement à Y les valeurs

$$h(\overline{x}, 0), h(\overline{x}, 1), \ldots, h(\overline{x}, x_{n+1})$$

puis s'arrête.

2ème cas : $h(\overline{x}, x_{n+1})$ n'est est pas défini. Si $f(\overline{x})$ n'est pas défini, alors l'exécution de la pseudo-instruction (1) de $\mathcal{P}$ ne s'arrête pas. Sinon, soit z_0 le plus petit entier naturel z tel que $g(\overline{x}, z, h(\overline{x}, z))$ n'est pas défini ($z_0 < x_{n+1}$) ; alors la (z_0+1)ème exécution de la pseudo-instruction (3) ne s'arrête pas.

■

2.1.3 Les fonctions primitives récursives

Définition 2.3 *On appelle* **fonctions de base** *les fonctions suivantes :*

- *la fonction $s \in \mathcal{F}_1$ appelée* **successeur** *et définie par $s(x) = x + 1$;*
- *pour chaque $p \geq 0$, la fonction $N_p \in \mathcal{F}_p$ qui associe 0 à tout p-uplet ;*
- *pour chaque $n \geq 1$, les fonctions $u_i^n \in \mathcal{F}_n$ $(1 \leq i \leq n)$ appelées* **projections** *et définies par $u_i^n(x_1, \ldots, x_n) = x_i$.*

Remarque 2.4 Toutes les fonctions de base sont des fonctions totales.

Notation. On note $\mathcal{F}^*$ l'ensemble des fonctions partielles d'arité quelconque :

$$\mathcal{F}^* = \bigcup_{n=0}^{\infty} \mathcal{F}_n.$$

Définition 2.4 *Soit $\mathcal{H}$ un sous-ensemble de $\mathcal{F}^*$ (c'est-à-dire un ensemble de fonctions partielles). On dit que $\mathcal{H}$ est* **stable par composition** *si pour toute fonction $f \in \mathcal{F}_k$ et toutes fonctions $g_1, \ldots, g_k \in \mathcal{F}_n$,*

$$f, g_1, \ldots, g_k \in \mathcal{H} \Rightarrow f(g_1, \ldots, g_k) \in \mathcal{H}.$$

On dit que $\mathcal{H}$ est **stable par récursion primitive** *si pour toute fonction $f \in \mathcal{F}_n$ et toute fonction $g \in \mathcal{F}_{n+2}$,*

$$f, g \in \mathcal{H} \Rightarrow rec(f, g) \in \mathcal{H}.$$

Il est évident qu'il existe des sous-ensembles de $\mathcal{F}^*$ contenant les fonctions de base et stables par composition et récursion primitive (par exemple $\mathcal{F}^*$ lui-même !), donc l'intersection de tous les sous-ensembles de $\mathcal{F}^*$ contenant les fonctions de base et stable par composition et récursion primitive est définie, c'est un sous-ensemble de $\mathcal{F}^*$ contenant les fonctions de base et stable par composition et récursion primitive, et il est inclus dans tout sous-ensemble de $\mathcal{F}^*$ contenant les fonctions de base et stable par composition et récursion primitive. C'est ce sous-ensemble de $\mathcal{F}^*$ qui constitue l'ensemble des **fonctions primitives récursives**.

Définition 2.5 *L'ensemble des* **fonctions primitives récursives** *est le plus petit (pour la relation d'inclusion) sous-ensemble de $\mathcal{F}^*$ contenant les fonctions de base et stable par composition et récursion primitive.*

Définition 2.6 *Soit P un prédicat d'arité quelconque. P est un* **prédicat primitif récursif** *si et seulement si sa fonction indicatrice est primitive récursive.*

La définition 2.5 donne une méthode pour démontrer que toutes les fonctions primitives récursives satisfont une certaine propriété.

Théorème 2.5 (Principe d'induction structurelle sur les fonctions primitives récursives) *Soit $\mathbb{P}$ une propriété des fonctions partielles. Pour montrer que toutes les fonctions primitives récursives satisfont $\mathbb{P}$, il suffit de montrer que*

1. *Toutes les fonctions de base satisfont* $\mathbb{P}$ *;*
2. *Si la fonction* $f \in \mathcal{F}_k$ *et toutes les fonctions* $g_1, \ldots, g_k \in \mathcal{F}_n$ *satisfont* $\mathbb{P}$*, alors la fonction* $f(g_1, \ldots, g_k)$ *satisfait* $\mathbb{P}$ *;*
3. *Si la fonction* $f \in \mathcal{F}_n$ *et la fonction* $g \in \mathcal{F}_{n+2}$ *satisfont* $\mathbb{P}$*, alors la fonction* $rec(f, g)$ *satisfait* $\mathbb{P}$*.* ∎

Proposition 2.6 *Toutes les fonctions primitives récursives sont totales.*

Démonstration : On utilise le principe d'induction du théorème 2.5 :
1. Les fonctions de base sont totales (cf. la remarque 2.4) ;
2. Le schéma de composition "préserve la totalité" (cf. la proposition 2.1) ;
3. Le schéma de récursion primitive "préserve la totalité" (cf. la proposition 2.3). ∎

Théorème 2.7 *Toutes les fonctions primitives récursives sont calculables.*

Démonstration : On utilise le principe d'induction du théorème 2.5 :
1. Les fonctions de base sont calculables (on laisse au lecteur le soin de vérifier ce point) ;
2. Le schéma de composition "préserve la calculabilité" (cf. le théorème 2.2) ;
3. Le schéma de récursion primitive "préserve la calculabilité" (cf. le théorème 2.4). ∎

On peut définir les fonctions primitives récursives d'une façon légèrement différente. On définit une suite $\mathcal{H}_0, \mathcal{H}_1, \ldots$ de sous-ensembles de $\mathcal{F}^*$:

1. $\mathcal{H}_0$ est l'ensemble des fonctions de base ;
2. Pour tout entier naturel n,

$$\mathcal{H}_{n+1} = \mathcal{H}_n \cup \{f(g_1, \ldots, g_n) \mid f, g_1, \ldots, g_n \in \mathcal{H}_n\} \cup \{\mathrm{rec}(f, g) \mid f, g \in \mathcal{H}_n\}.$$

   Autrement dit $\mathcal{H}_{n+1}$ est constitué des fonctions de $\mathcal{H}_n$ et des fonctions obtenues par composition ou récursion primitive à partir de fonctions de $\mathcal{H}_n$.

On a défini la suite $(\mathcal{H}_n)_{n \in \mathbb{N}}$ par **récurrence sur** n. Il va de soi que cette suite est **croissante pour la relation d'inclusion** : pour tout entier naturel n,

$$\mathcal{H}_n \subseteq \mathcal{H}_{n+1}.$$

Les fonctions de $\mathcal{H}_n$ sont les fonctions obtenues à partir des fonctions initiales par **au plus** n **applications des schémas de composition ou de récursion**.

Soit $\mathcal{H}^* = \bigcup_{n=0}^{\infty} \mathcal{H}_n$.

Proposition 2.8 *$\mathcal{H}^*$ est égal à l'ensemble des fonctions primitives récursives.*

Démonstration : On montre que toutes les fonctions primitives récursives appartiennent à $\mathcal{H}^*$ puis que toutes les fonctions de $\mathcal{H}^*$ sont primitives récursives.

1. $\mathcal{H}^*$ contient les fonctions de base (car $\mathcal{H}_0 \subseteq \mathcal{H}^*$!)

$\mathcal{H}^*$ est stable par composition ; en effet, soit $f \in \mathcal{F}_n$ et $g_1, \ldots, g_k \in \mathcal{F}_n$ des fonctions de $\mathcal{H}^*$. Alors il existe un entier naturel m tel que toutes les fonctions $f, g_1, \ldots, g_k$ appartiennent à $\mathcal{H}_m$ (par croissance de la suite $(\mathcal{H}_n)_{n\in\mathbb{N}}$), et $f(g_1, \ldots, g_k)$ appartient à $\mathcal{H}_{m+1}$, donc à $\mathcal{H}^*$.

$\mathcal{H}^*$ est stable par récursion primitive ; en effet, soit $f \in \mathcal{F}_n$ et $g \in \mathcal{F}_{n+2}$ des fonctions de $\mathcal{H}^*$. Alors il existe un entier naturel m tel que toutes les fonctions f et g appartiennent à $\mathcal{H}_m$ (par croissance de la suite $(\mathcal{H}_n)_{n\in\mathbb{N}}$), et $\mathrm{rec}(f, g)$ appartient à $\mathcal{H}_{m+1}$, donc à $\mathcal{H}^*$.

Comme l'ensemble des fonctions primitives récursives est inclus dans tous les sous-ensembles de $\mathcal{F}^*$ contenant les fonctions de base et stables par composition et récursion primitive, on en conclut que toutes les fonctions primitives récursives appartiennent à $\mathcal{H}^*$.

2. L'ensemble des fonctions primitives récursives contient les fonctions de base, donc $\mathcal{H}_0$. Pour tout entier naturel n, si l'ensemble des fonctions primitives récursives contient $\mathcal{H}_n$, il contient $\mathcal{H}_{n+1}$, car il est stable par composition et récursion primitive. On vient de montrer **par récurrence sur** n que l'ensemble des fonctions primitives récursives contient tous les ensembles $\mathcal{H}_n$, donc leur union, c'est-à-dire $\mathcal{H}^*$. ■

Cela mène à une nouvelle définition.

Définition 2.7 *L'ensemble des fonctions primitives récursives est l'ensemble des fonctions obtenues à partir des fonctions de base par* **un nombre fini d'applications des schémas de composition ou de récursion primitive**.

C'est la définition 2.7 qui donne la méthode pour montrer qu'une fonction totale donnée est primitive récursive.

Exemple 2.5 Toutes les fonctions constantes sont primitives récursives. En effet, pour tout entier $p \geq 0$, la fonction $N_p \in \mathcal{F}_p$ qui associe 0 à tout p-uplet est une fonction de base et si $c^p - k$ désigne la fonction de $\mathcal{F}_p$ qui associe k à tout p-uplet, on a $c^p_0 = N_p$ et $c^p_{k+1} = s \circ c^p_k$.

la fonction constante unaire égale à 0 est une fonction de base et pour tout entier naturel k, si f_k désigne la fonction unaire constante égale à k, $f_{k+1} = s \circ f_k$. Enfin, si g_k désigne la fonction de $\mathcal{F}_n$ constante égale à k, $g_k = f_k \circ u^n_1$.

Exemple 2.6 L'addition est primitive récursive. On peut l'obtenir par récursion primitive à partir de u^1_1 et de $g = s \circ u^3_3$:

$$\begin{cases} x+0 &= x &= u^1_1(x) \\ x+(t+1) &= (x+t)+1 &= s(u^3_3(x,t,x+t)). \end{cases}$$

On en déduit que la multiplication est primitive récursive, puisqu'elle est obtenue par récursion primitive à partir de N_1, qui est une fonction de base, et de $\mathrm{Add}(u^3_3, u^3_1)$ où Add désigne l'addition (cf. l'exemple 2.4 ci-dessus.)

Il est facile d'en déduire que les fonctions $x \mapsto k \cdot x$, où k est un entier naturel fixé, sont primitives récursives.

Exemple 2.7 En posant $0^0 = 1$, l'exponentiation $\exp(x,y) = x^y$ est primitive récursive car

$$\begin{cases} \exp(x,0) &= 1 \\ \exp(x,t+1) &= x^t \cdot x &= \mathrm{Mult}(u^3_3(x,t,\exp(x,t)), u^3_1(x,t,\exp(x,t))). \end{cases}$$

Il est facile d'en déduire que les fonctions $x \mapsto x^k$, où k est un entier naturel fixé, sont primitives récursives.

Exemple 2.8 La fonction unaire **prédécesseur** définie par

$$p(x) = \begin{cases} x-1 & \text{si } x \geq 1 \\ 0 & \text{si } x = 0 \end{cases}$$

est primitive récursive, puisqu'elle est obtenue par récursion primitive à partir de 0 et de u^2_1.

$$\begin{cases} p(0) &= 0 \\ p(t+1) &= t &= u^2_1(t,p(t)). \end{cases}$$

Exemple 2.9 Des exemples 2.2, 2.6 et 2.7, il n'est pas possible de déduire directement que la fonction $x \mapsto 4x^2 - 2x$ est primitive récursive car la fonction Moins utilisée dans la composition de l'exemple 2.2 n'est pas primitive récursive (elle n'est pas totale!). Toutefois, on peut lui substituer la fonction

$$x_1 \dot{-} x_2 = \begin{cases} (x_1 - x_2) & \text{si } x_1 \geq x_2 \\ 0 & \text{sinon} \end{cases}$$

qui est primitive récursive car

$$\begin{cases} x \dot{-} 0 & = & x & = & u_1^1(x) \\ x \dot{-} (t+1) & = & p(x \dot{-} t) & = & p(u_3^3(x, t, x \dot{-} t)). \end{cases}$$

D'ailleurs, toutes les fonctions polynômes à valeurs positives ou nulles sont primitives récursives.

Exemple 2.10 Les deux fonctions $\alpha, \overline{\alpha} \in \mathcal{F}_1$ définies par

$$\alpha(x) = \begin{cases} 1 & \text{si } x = 0 \\ 0 & \text{sinon} \end{cases} \quad \text{et} \quad \overline{\alpha}(x) = \begin{cases} 0 & \text{si } x = 0 \\ 1 & \text{sinon} \end{cases}$$

sont très utiles. Il est facile de vérifier qu'elles sont primitives récursives.

2.1.4 Autres schémas

Proposition 2.9 (Définition par cas) *Soit $g_1, \ldots, g_m, f$ des fonctions de $\mathcal{F}_n$ et $P_1, \ldots, P_m$ $(m \geq 1)$ des prédicats n-aires* **deux à deux disjoints** *(*i.e. *$P_i \cap P_j = \emptyset$ dès que $i \neq j$). On définit la fonction h de la façon suivante :*

$$h(x_1, \ldots, x_n) = \begin{cases} g_1(x_1, \ldots, x_n) & si & P_1(x_1, \ldots, x_n) \\ \vdots & & \vdots \\ g_m(x_1, \ldots, x_n) & si & P_m(x_1, \ldots, x_n) \\ f(x_1, \ldots, x_n) & sinon & \end{cases}$$

Si les fonctions $g_1, \ldots, g_m, f$ et les prédicats $P_1, \ldots, P_m$ sont primitifs récursifs, alors la fonction h est primitive récursive.

Démonstration : Il suffit de remarquer que

$$f(x_1, \ldots, x_n) = \sum_{i=1}^{m} g_i(x_1, \ldots, x_n) \cdot \mathbf{1}_{P_i}(x_1, \ldots, x_n) \\ + f(x_1, \ldots, x_n) \cdot \mathbf{1}_{\neg P_1 \wedge \cdots \wedge \neg P_m}(x_1, \ldots, x_n). \quad \blacksquare$$

Exemple 2.11 Le schéma de définition par cas permet d'établir que les fonctions $\delta, \overline{\delta} \in \mathcal{F}_2$ définies par

$$\delta(x_1, x_2) = \begin{cases} 1 & \text{si } x_1 = x_2 \\ 0 & \text{sinon} \end{cases} \quad \text{et} \quad \overline{\delta}(x_1, x_2) = \begin{cases} 0 & \text{si } x_1 = x_2 \\ 1 & \text{sinon} \end{cases}$$

sont primitives récursives.

Proposition 2.10 (Somme et produit bornés) *Soit f une fonction de $\mathcal{F}_{n+1}$. On définit les fonctions g et h de la façon suivante :*

$$\begin{array}{rrcl} & g(x_1,\ldots,x_n,y) & = & \sum_{t=0}^{y} f(x_1,\ldots,x_n,t) \\ \textit{et} & h(x_1,\ldots,x_n,y) & = & \prod_{t=0}^{y} f(x_1,\ldots,x_n,t) \end{array}$$

Si la fonction f est primitive récursive, alors les fonctions g et h sont primitives récursives.

Démonstration : On montre que g et h s'obtiennent par récursion primitive à partir de f :

$$\left\{ \begin{array}{lcl} g(x_1,\ldots,x_n,0) & = & f(x_1,\ldots,x_n,0) \\ g(x_1,\ldots,x_n,t+1) & = & g(x_1,\ldots,x_n,t) + f(x_1,\ldots,x_n,t+1) \end{array} \right.$$

$$\left\{ \begin{array}{lcl} h(x_1,\ldots,x_n,0) & = & f(x_1,\ldots,x_n,0) \\ h(x_1,\ldots,x_n,t+1) & = & h(x_1,\ldots,x_n,t) \cdot f(x_1,\ldots,x_n,t+1) \end{array} \right.$$

■

Remarque 2.5 la proposition 2.10 reste vraie si on fait commencer la somme et le produit définissant respectivement g et h à 1 au lieu de 0 ; il suffit de modifier les schémas de récursion primitive de la démonstration ci-dessus en posant

$$\begin{array}{rrcl} & g(x_1,\ldots,x_n,0) & = & 0 \\ \text{et} & h(x_1,\ldots,x_n,0) & = & 1 \end{array}$$

Définition 2.8 (Schéma de minimisation bornée) *Soit f une fonction* **totale** *de $\mathcal{F}_{n+1}$. La* **fonction obtenue par minimisation bornée à partir de** *f est la fonction h de $\mathcal{F}_{n+1}$ définie par*

- *$h(x_1,\ldots,x_n,y)$ est le plus petit $z \leq y$ tel que $f(x_1,\ldots,x_n,z) = 0$, si un tel z existe ;*
- *$h(x_1,\ldots,x_n,y) = y+1$ sinon.*

Notation. On note $\mu_b f$ la fonction h de la définition 2.8, obtenue par minimisation bornée à partir de f. L'entier $h(x_1,\ldots,x_n,y)$ est noté $\mu_b f(x_1,\ldots,x_n,y)$ ou $\min\limits_{z\leq y}[f(x_1,\ldots,x_n,z)=0]$.

Soit P un prédicat $(n+1)$-aire. On note $\mu_b P$ la fonction $\mu_b \mathbf{1}_{\neg P}$, où $\mathbf{1}_{\neg P}$ est la fonction indicatrice de $\neg P$, qui satisfait $\mathbf{1}_{\neg P}(x_1,\ldots,x_n,z) =$

0 si $P(x_1, \ldots, x_n, z)$. On note alors $\min_{z \leq y}[P(x_1, \ldots, x_n, z)]$ l'entier naturel $\mu_b P(x_1, \ldots, x_n, y)$.

Proposition 2.11 *Si la fonction $f \in \mathcal{F}_{n+1}$ est primitive récursive, alors la fonction $h \in \mathcal{F}_{n+1}$ obtenue par minimisation bornée à partir de f est primitive récursive.*

Démonstration : Voici une stratégie pour définir h par récursion primitive. Pour calculer $h(x_1, \ldots, x_n, y)$, on commence par additionner les valeurs

$$\overline{\alpha}(f(x_1, \ldots, x_n, 0)), \ldots, \overline{\alpha}(f(x_1, \ldots, x_n, y))$$

Si la somme vaut $y + 1$, alors toutes les valeurs

$$f(x_1, \ldots, x_n, 0), \ldots, f(x_1, \ldots, x_n, y)$$

sont différentes de 0. Dans ce cas, on ajoute 1 à la valeur "précédente" de h.

Si la somme est plus petite que $y + 1$, c'est qu'il existe un $z \leq y$ tel que $f(x_1, \ldots, x_n, z) = 0$, et on conserve la valeur "précédente" de h.

Formellement, cela s'écrit :

$$\left\{ \begin{array}{lcl} h(x_1, \ldots, x_n, 0) & = & \left\{ \begin{array}{ll} 0 & \text{si } f(x_1, \ldots, x_n, 0) = 0 \\ 1 & \text{sinon} \end{array} \right. \\ h(x_1, \ldots, x_n, t+1) & = & h(x_1, \ldots, x_n, t) + \delta(\sum_{i=0}^{t+1} \overline{\alpha}(f(x_1, \ldots, x_n, i)), t + 2). \end{array} \right.$$

On remarque que $h(x_1, \ldots, x_n, 0)$ est lui-même défini par cas dans le schéma de récursion primitive définissant $h(x_1, \ldots, x_n, t)$. ■

Exemple 2.12 On peut montrer que la fonction $q \in \mathcal{F}_2$ définie par

$$\left\{ \begin{array}{ll} q(x_1, x_2) = 0 & \text{si } x_2 = 0 \\ q(x_1, x_2) = x_1 \ \mathbf{div} \ x_2 & \text{sinon} \end{array} \right.$$

est primitive récursive en combinant schéma de minimisation bornée et schéma de définition par cas. En effet,

$$q(x_1, x_2) = \left\{ \begin{array}{ll} 0 & \text{si } x_2 = 0 \\ q'[u_2^2, u_1^2](x_1, x_2) & \text{sinon} \end{array} \right.$$

avec $q'(x_1, x_2) = \min_{t \leq x_2}[(t + 1) \cdot x_1 > x_2]$.

2.2 Les fonctions récursives

2.2.1 La fonction d'Ackermann

Toutes les fonctions primitives récursives sont calculables et totales. Dans le premier chapitre, on a vu des fonctions calculables non totales (comme la fonction nulle part définie, par exemple !) Les fonctions primitives récursives constituent donc un sous-ensemble strict des fonctions calculables. Il existe même des fonctions totales calculables qui ne sont pas primitives récursives :

Exemple 2.13 La **fonction d'Ackermann** $\xi \in \mathcal{F}_2$ est définie de la façon suivante :

$$\left\{\begin{array}{ll} \text{Pour tout entier } x & \xi(0,y) = 2^y \\ \text{Pour tout entier } y & \xi(x,0) = 1 \\ \text{Pour tous entiers } x \text{ et } y & \xi(x+1,y+1) = \xi(x,\xi(x+1,y)) \end{array}\right.$$

Pour tout entier naturel n, on définit la fonction $\xi_n \in \mathcal{F}_1$ par

$$\xi_n(y) = \xi(n,y).$$

On remarque que $\xi_0(y) = 2^y$ et que pour tout entier naturel n, ξ_{n+1} est définie par récursion primitive à partir de ξ_n :

$$\left\{\begin{array}{lcl} \xi_{n+1}(0) & = & 1 \\ \xi_{n+1}(t+1) & = & \xi_n \circ u_2^2(t,\xi_{n+1}(t)). \end{array}\right.$$

Cela montre qu'il existe une et une seule fonction de $\mathcal{F}_2$ satisfaisant les conditions de la définition de ξ (cette fonction est évidemment totale) et que toutes les fonctions ξ_n sont primitives récursives (il suffit de faire une **récurrence sur** n). D'autre part, on peut effectivement calculer la fonction ξ à l'aide d'un programme avec branchement. En revanche, rien ne nous permet d'affirmer que la fonction ξ est primitive récursive ; on peut d'ailleurs démontrer qu'elle ne l'est pas !

Théorème 2.12 *La fonction d'Ackermann n'est pas primitive récursive.*

Démonstration : La démonstration consiste à montrer que la fonction d'Ackermann croît plus vite que toute fonction primitive récursive. ■

2.2.2 Le schéma de minimisation

Définition 2.9 (Schéma de minimisation non bornée). *Soit f une fonction de $\mathcal{F}_{n+1}$. La fonction obtenue par minimisation (non bornée) à partir de f est la fonction h de $\mathcal{F}_n$ définie de la façon suivante :*

- *$h(x_1, \ldots, x_n)$ est le plus petit y tel que $f(x_1, \ldots, x_n, z)$ est défini pour tout $z \leq y$ et $f(x_1, \ldots, x_n, y) = 0$, si un tel y existe ;*
- *$h(x_1, \ldots, x_n)$ n'est pas défini sinon.*

Notation. On note μf la fonction h de la définition 2.9, obtenue par minimisation non bornée à partir de f.

$h(x_1, \ldots, x_n)$ est noté $\mu f(x_1, \ldots, x_n, y)$ ou $\min\limits_z[f(x_1, \ldots, x_n, z) = 0]$.

Soit P un prédicat $(n+1)$-aire. On note μP la fonction $\mu \mathbf{1}_{\neg P}$.

$\mu P(x_1, \ldots, x_n, y)$ est aussi noté $\min\limits_z[P(x_1, \ldots, x_n, z)]$.

Exemple 2.14 $$\text{Moins}(x_1, x_2) = \min_z[x_2 + z = x_1].$$

Remarque 2.6 À la différence de tous les schémas présentés auparavant, le schéma de minimisation non bornée permet de construire une fonction h non totale à partir d'une fonction f totale.

Théorème 2.13 *Soit f une fonction de $\mathcal{F}_{n+1}$ et h la fonction de $\mathcal{F}_n$ obtenue par minimisation non bornée à partir de f. Si f est calculable, alors h est calculable.*

Démonstration : S'il existe un programme avec branchement calculant la fonction f, alors le pseudo-programme suivant calcule h :

$$
\begin{array}{ll}
[A] & Z \leftarrow f(X_1, \ldots, X_n, Y) \\
 & \text{IF } Z = 0 \text{ GOTO } E \\
 & Y \leftarrow Y + 1 \\
 & \text{GOTO } A
\end{array}
$$

■

2.2.3 Les fonctions partielles récursives

Définition 2.10 *Soit $\mathcal{H}$ un sous-ensemble de $\mathcal{F}^*$ (c'est-à-dire un ensemble de fonctions partielles). On dit que $\mathcal{H}$ est* **stable par minimisation** *si pour toute fonction $f \in \mathcal{F}_{n+1}$,*

$$f \in \mathcal{H} \Rightarrow \mu f \in \mathcal{H}.$$

Définition 2.11 *L'ensemble des fonctions partielles récursives est le plus petit ensemble contenant les fonctions de base et stable par composition, par récursion primitive et par minimisation non bornée.*

Définition 2.12 *Un prédicat n-aire est récursif si sa fonction caractéristique est (totale) récursive.*

La définition 2.11 donne une méthode pour démontrer que toutes les fonctions récursives satisfont une certaine propriété.

Théorème 2.14 (Principe d'induction structurelle sur les fonctions récursives) *Soit $\mathbb{P}$ une propriété des fonctions partielles. Pour montrer que toutes les fonctions récursives satisfont $\mathbb{P}$, il suffit de montrer que*

1. *Toutes les fonctions de base satisfont $\mathbb{P}$;*
2. *Si la fonction $f \in \mathcal{F}_k$ et toutes les fonctions $g_1, \ldots, g_k \in \mathcal{F}_n$ satisfont $\mathbb{P}$, alors la fonction $f(g_1, \ldots, g_k)$ satisfait $\mathbb{P}$;*
3. *Si la fonction $f \in \mathcal{F}_n$ et la fonction $g \in \mathcal{F}_{n+2}$ satisfont $\mathbb{P}$, alors la fonction $rec(f, g)$ satisfait $\mathbb{P}$.*
4. *Si la fonction $f \in \mathcal{F}_{n+1}$ satisfait $\mathbb{P}$, alors la fonction μf satisfait $\mathbb{P}$.* ■

Dans le paragraphe 2.1.3, on a défini de deux manières différentes les fonctions primitives récursives. Un raisonnement similaire conduit directement à :

Définition 2.13 *L'ensemble des fonctions partielles récurives est l'ensemble des fonctions obtenues à partir des fonctions de base par* **un nombre fini d'applications des schémas de composition, de récursion primitive ou de minimisation non bornée**.

Exemple 2.15 La plupart des (pour ne pas dire toutes les) fonctions "usuelles" étant primitives récursives, on est en peine de donner un exemple concret de fonction (totale) récursive qui ne soit pas primitive récursive. La fonction d'Ackermann ξ est une telle fonction (c'est assez difficile à montrer directement) ; en revanche, son graphe, c'est-à-dire le prédicat ternaire A défini par

$$A(x_1, x_2, x_3) \Leftrightarrow x_3 = \xi(x_1, x_2)$$

est primitif récursif !

Théorème 2.15 *Toutes les fonctions partielles récursives sont calculables.*

Démonstration : On utilise le principe d'induction du théorème 2.14 :

1. Les fonctions de base sont calculables ;
2. Le schéma de composition "préserve la calculabilité" (cf. le théorème 2.2) ;
3. Le schéma de récursion primitive "préserve la calculabilité" (cf. le théorème 2.4) ;
4. Le schéma de minimisation non bornée "préserve la calculabilité" (cf. le théorème 2.13). ■

Remarque 2.7 Dans le chapitre 3, on montrera que, réciproquement, toutes les fonctions partielles calculables sont récursives.

Chapitre 3

Universalité

Dans ce chapitre, on présente deux importants résultats qui permettent de préciser les contours de la notion de calculabilité : d'une part, il existe des prédicats et fonctions partielles "naturels" qui ne sont pas calculables (indécidabilité du problème de l'arrêt), d'autre part, il existe une fonction partielle calculable de $\mathcal{F}_{n+1}$ qui énumère toutes les fonctions partielles calculables de $\mathcal{F}_n$ (Théorème d'universalité). Ces deux résultats reposant en grande partie sur la possibilité de coder "effectivement" les programmes, c'est par cet aspect qu'on commence.

3.1 Codage des programmes par des entiers

3.1.1 Codage des paires d'entiers naturels

On cherche à numéroter (à partir de 0) les couples d'entiers (x_1, x_2) à l'aide d'une bijection f en parcourant $\mathbb{N} \times \mathbb{N}$ de la façon suivante (l'entier naturel qui setrouve à l'intersection de la ligne i et de la colonne j étant le numéro du couple (i, j)) :

$$\begin{array}{c|cccccc}
 & 0 & 1 & 2 & 3 & 4 & \cdots \\
\hline
0 & 0 & 2 & 5 & 9 & \nearrow & \\
1 & 1 & 4 & 8 & \nearrow & & \\
2 & 3 & 7 & \nearrow & & & \\
3 & 6 & \nearrow & & & & \\
\vdots & \nearrow & & & & &
\end{array}$$

On remarque que si les couples (x_1, x_2) et (x'_1, x'_2) se trouvent sur la même diagonale , alors

$$x_1 + x_2 = x'_1 + x'_2.$$

On appellera donc $i$ème diagonale la diagonale constituée des couples (x_1, x_2) tels $x_1 + x_2 = i$.

Proposition 3.1 *Ce parcours de $\mathbb{N} \times \mathbb{N}$ selon les diagonales correspond à la fonction f définie par :*

$$f(x_1, x_2) = x_2 + \sum_{k=1}^{x_1+x_2} k.$$

L'usage est de noter $\langle \cdot,\cdot \rangle$ cette fonction ; on écrit donc $\langle x_1, x_2 \rangle$ au lieu de $f(x_1, x_2)$.

Démonstration : En effet, pour obtenir $\langle x_1, x_2 \rangle$, il faut compter les couples qui le précèdent dans l'énumération $\langle \cdot,\cdot \rangle$.

(x_1, x_2) se trouve en $x_2$ème position sur la (x_1+x_2)ème diagonale ; comme la $n$ème diagonale contient $n + 1$ couples, on a :

$$\langle x_1, x_2 \rangle = 1 + 2 + \cdots + (x_1+x_2) + x_2.$$

■

Remarque 3.1 Il est facile de vérifier que la fonction $\langle \cdot,\cdot \rangle$ est une bijection de $\mathbb{N} \times \mathbb{N}$ dans $\mathbb{N}$.

Proposition 3.2 *La fonction $\langle \cdot,\cdot \rangle$ est primitive récursive.*

Démonstration : On a :

$$\left\{ \begin{array}{lcl} \langle x, 0 \rangle & = & x(x+1)/2 \\ \langle x, t+1 \rangle & = & \langle x, t \rangle + x + t + 2 \\ & = & \mathrm{Add}(u_3^3(x, t, \langle x, t \rangle), \mathrm{Add}(\mathrm{Add}(u_1^3(x, t, \langle x, t \rangle), u_2^3(x, t, \langle x, t \rangle)), 2)) \end{array} \right.$$

■

Lemme 3.3 *Soit la fonction $cd \in \mathcal{F}_1$ définie par : $cd(n)$ est le numéro de la diagonale sur laquelle se trouve le $n$ème couple d'entiers (c'est-à-dire le couple (x_1, x_2) tel que $\langle x_1, x_2 \rangle = n$.)*

La fonction cd est primitive récursive.

Démonstration : On remarque que les couples de numéros n et $n+1$ se trouvent sur deux diagonales distinctes si et seulement si le couple de numéro $n+1$ se trouve au tout début de la diagonale $\mathrm{cd}(n)+1$. On en tire :

$$\begin{cases} \mathrm{cd}(0) & = & 0 \\ \mathrm{cd}(n+1) & = & \mathrm{cd}(n) + ((n+2) \dot{-} \langle \mathrm{cd}(n)+1, 0\rangle) \end{cases}$$

■

Proposition 3.4 *Les fonctions π_1 et π_2 définies par*

$$\pi_1(\langle x_1, x_2\rangle) = x_1 \qquad et \qquad \pi_2(\langle x_1, x_2\rangle) = x_2$$

sont primitives récursives.

Démonstration : $\pi_2(n)$ est la position du $n$ème couple sur la $\mathrm{cd}(n)$ème diagonale, d'où :

$$\pi_2(n) = n \dot{-} \langle \mathrm{cd}(n), 0\rangle$$

car $\langle \mathrm{cd}(n), 0\rangle$ est le premier couple figurant sur la diagonale où se trouve le $n$ème couple.

Comme $\pi_1(n) + \pi_2(n) = \mathrm{cd}(n)$, on a :

$$\pi_1(n) = \mathrm{cd}(n) - \pi_2(n).$$

■

Cela nous assure que le codage $\langle\cdot,\cdot\rangle$ est “effectif” dans les deux sens, puisque “codage” et “décodage” se définissent par des fonctions primitives récursives.

3.1.2 Codage des suites finies d'entiers naturels

On vient de voir comment coder effectivement les couples d'entiers naturels. Peut-on en faire autant avec les suites finies d'entiers naturels ? Commençons par définir la fonction $\tau_3 \in \mathcal{F}_3$ par :

$$\tau_3(x_1, x_2, x_3) = \langle x_1, \langle x_2, x_3\rangle\rangle.$$

La fonction τ_3 est primitive récursive (c'est la composée de fonctions primitives récursives !) ; elle est bijective (pour la même raison !) et si on lui associe les trois fonctions $\gamma_1^3, \gamma_2^3, \gamma_3^3 \in \mathcal{F}_1$ définies par

$$\begin{array}{rcl} \gamma_1^3(x) & = & \pi_1(x) \\ \gamma_2^3(x) & = & \pi_1 \circ \pi_2(x) \\ \gamma_3^3(x) & = & \pi_2 \circ \pi_2(x) \end{array}$$

on obtient un codage "effectif" des triplets d'entiers naturels.

On peut généraliser le procédé ci-dessus et définir pour chaque $n \geq 1$ une fonction $\tau_n \in \mathcal{F}_n$ et n fonctions $\gamma_1^n, \ldots, \gamma_n^n \in \mathcal{F}_1$, toutes bijectives et primitives récursives. On pose

$$\tau_1(x) = x \quad \text{et} \quad \gamma_1^1(x) = x$$

puis

$$\tau_2(x_1, x_2) = \langle x_1, x_2 \rangle \quad \text{et} \quad \gamma_1^2 = \pi_1, \ \ \gamma_2^2 = \pi_2$$

et, pour $n \geq 2$,

$$\tau_{n+1}(x_1, \ldots, x_{n-1}, x_n, x_{n+1}) = \tau_n(x_1, \ldots, x_{n-1}, \tau_2(x_n, x_{n+1}))$$
$$\text{et} \quad \left\{ \begin{array}{l} \gamma_1^{n+1} = \gamma_1^n, \ \ldots, \ \gamma_{n-1}^{n+1} = \gamma_{n-1}^n \\ \gamma_n^{n+1} = \gamma_1^2 \circ \gamma_n^n, \ \ \gamma_{n+1}^{n+1} = \gamma_2^2 \circ \gamma_n^n \end{array} \right.$$

On obtient ainsi, pour chaque $n \geq 1$, un codage "effectif" des n-uplets d'entiers naturels.

Définition 3.1 *Soit $\mathbb{N}^{<\omega} = \bigcup_{k=1}^{\infty} \mathbb{N}^k$ l'ensemble des suites finies non vides d'entiers naturels. On définit maintenant une fonction τ de $\mathbb{N}^{<\omega}$ dans $\mathbb{N}$:*

$$\tau(x_1, \ldots, x_n) = \langle n, \tau_n(x_1, \ldots, x_n) \rangle$$

Attention, τ n'est pas une fonction de $\mathcal{F}^*$! On ne peut donc pas dire *stricto sensu* que τ est primitive récursive.

Proposition 3.5 *τ est une bijection de $\mathbb{N}^{<\omega}$ dans un ensemble primitif récursif. Autrement dit, τ est injective et son image est un prédicat unaire primitif récursif.*

Démonstration : Soit σ et σ' deux suites d'entiers naturels qui satisfont $\tau(\sigma) = \tau(\sigma') = m$; alors elles ont la même longueur $n = \pi_1(m)$ et sont égales car $\tau_n(\sigma) = \tau_n(\sigma') = \pi_2(m)$.

En outre, l'image de τ est $\{x \mid \pi_1(x) \neq 0\}$, qui est primitif récursif. ■

On peut donc "décoder effectivement", car on peut décider de façon primitive récursive si un entier naturel m est le code (par τ) d'une suite non vide d'entiers naturels (cf. la proposition 3.5), puis calculer la longueur $\pi_1(m)$ de la suite codée par m et le $i$ème élément $\gamma_i^{\pi_1(m)} \circ \pi_2(m)$ de la suite codée par m. D'ailleurs, on remarque que :

Proposition 3.6 *La fonction φ de $\mathcal{F}_3$ définie par*

$$\varphi(n,i,x) = \begin{cases} \gamma_i^n(x) & si\ 1 \leq i \leq n \\ 0 & sinon \end{cases}$$

est primitive récursive.

Démonstration : On montre d'abord que la fonction $\psi \in \mathcal{F}_2$ définie par

$$\psi(x_1,x_2) = \varphi(x_1,x_1,x_2)$$

est primitive récursive. En effet,

$$\begin{cases} \psi(0,y) & = & 0 \\ \psi(t{+}1,y) & = & \begin{cases} y & \text{si } t=0 \\ \pi_2(\psi(t,y)) & \text{sinon} \end{cases} \end{cases}$$

D'où

$$\begin{array}{lcl} \varphi(0,i,x) & = & 0 \\ \varphi(1,i,x) & = & \begin{cases} x & \text{si } i=1 \\ 0 & \text{sinon} \end{cases} \\ \varphi(n{+}1,i,x) & = & \begin{cases} 0 & \text{si } i=0 \text{ ou } i>n+1 \\ \varphi(n,i,x) & \text{si } 0<i<n \\ \pi_1(\varphi(n,i,x)) & \text{si } n>0 \text{ et } i=n \\ \psi(n+1,x) & \text{si } n>0 \text{ et } i=n+1 \end{cases} \end{array}$$

■

3.1.3 Codage des instructions

On commence par affecter à chaque variable V l'entier naturel $\#V$ qui est son numéro d'ordre (ici on compte à partir de 1 !) dans l'énumération

$$Y\ X_1\ Z_1\ X_2\ Z_2\ X_3\ Z_3\ \ldots$$

et à chaque étiquette L l'entier naturel $\#L$ qui est son numéro d'ordre (on compte encore à partir de 1 !) dans l'énumération

$$A_1\ B_1\ C_1\ D_1\ E_1\ A_2\ B_2\ C_2\ D_2\ E_2\ A_3\ \ldots$$

Définition 3.2 *Soit maintenant I une instruction du langage de programmes $\mathcal{S}$. On lui associe l'entier naturel $\#I$ défini par*

$$\#I = \tau_3(a, b, c) = \langle a, \langle b, c\rangle\rangle$$

où les entiers naturels a, b, c sont donnés par

1. *$a = 0$ si I n'est pas étiquetée ; $a = \#L$ si I est étiquetée par L ;*
2. *Si V est la variable qui apparaît dans I, alors $c = \#V - 1$;*
3. *$b = 0$ si I est de la forme $V \leftarrow V + 1$;*
4. *$b = 1$ si I est de la forme $V \leftarrow V - 1$;*
5. *$b = \#L' + 1$ si I est de la forme IF $V \neq 0$ GOTO L'.*

Exemple 3.1 Soit I l'instruction

$$[A] \quad \text{IF } X \neq 0 \text{ GOTO } B$$

Alors $\#I = \langle 1, \langle 2{+}1, 2{-}1\rangle\rangle = \langle 1, 11\rangle = 89$

Le paragraphe 3.1.2 permet d'affirmer :

Proposition 3.7 *Le codage des instructions selon $\#$ est effectif.* ∎

Proposition 3.8 *Le décodage est également effectif. À chaque entier naturel m correspond une et une seule instruction I telle que $\#I = m$, et cette instruction peut être effectivement reconstruite à partir de m.*

Démonstration : On suit la méthode suivante :

1. Si $\pi_1(m) = 0$, alors I n'est pas étiquetée ; si $\pi_1(m) \neq 0$, alors I est étiquetée par la $\pi_1(m)$ème étiquette de l'énumération ;
2. La variable V qui apparaît dans I occupe la $i$ème place dans l'énumération, où $i = \pi_2 \circ \pi_2(m) + 1$;
3. Enfin, I est de la forme

$$\begin{array}{lll}
V \leftarrow V + 1 & \text{si} & \pi_1 \circ \pi_2(m) = 0 \\
V \leftarrow V - 1 & \text{si} & \pi_1 \circ \pi_2(m) = 1 \\
\text{IF } V \neq 0 \text{ GOTO } L & \text{si} & j = \pi_1 \circ \pi_2(m) - 1 > 0 \text{ et} \\
 & & L \text{ est la } j\text{ème étiquette de l'énumération}
\end{array}$$

∎

3.1.4 Codage des programmes

Soit maintenant $\mathcal{P}$ un programme non vide de $\mathcal{S}$, égal à la suite d'instructions $I_1, I_2, \ldots, I_k$. On pose

$$\#\mathcal{P} = \tau(\#I_1, \#I_2, \ldots, \#I_k).$$

De nouveau, les développements du paragraphe 3.1.2 (les fonctions τ et φ) permettent de conclure qu'on a là un codage effectif (dans les deux sens !)

Remarque 3.2 On comprend maintenant pourquoi on a prévu d'abandonner la contrainte interdisant les programmes dont plusieurs instructions portent la même étiquette (cf. la note de la définition 1.4 et celle de la définition 1.7). En effet, cette contrainte, satisfaisante quand il s'agit d'écrire des programmes, rend l'identification des codes des programmes "légaux" assez pénible !

Remarque 3.3 On a laissé de côté le programme vide dans le codage qu'on vient de présenter. Il suffit de le coder par 0, qui n'est pas utilisé pour coder les programmes non vides.

3.2 Le problème de l'arrêt

On définit un prédicat binaire Arrêt de la façon suivante : soit (x_1, x_2) un couple d'entiers naturels et $\mathcal{P}$ le programme de $\mathcal{S}$ tel que $\#(\mathcal{P}) = x_2$,
- $(x_1, x_2) \in$ Arrêt si $\psi_{\mathcal{P}}^{(1)}(x_1)$ est défini ;
- $(x_1, x_2) \notin$ Arrêt si $\psi_{\mathcal{P}}^{(1)}(x_1)$ n'est pas défini.

Autrement dit,

$$\text{Arrêt}(x_1, x_2) \Leftrightarrow \text{ le programme de numéro } x_2 \text{ s'arrête sur l'entrée } x_1$$

Théorème 3.9 (Indécidabilité[1] du problème de l'arrêt) *Le prédicat Arrêt n'est pas calculable.*

Démonstration : Supposons que Arrêt soit calculable. Alors on peut construire le programme $\mathcal{P}$ qui est l'expansion de la macro suivante :

[A] IF Arrêt(X,X) GOTO A

[1] Il s'agit d'une des acceptions du vocable en informatique.

On remarque que :

$$\psi_{\mathcal{P}}^{(1)}(x) = \begin{cases} \uparrow & \text{si} \quad (x,x) \in \text{Arrêt} \\ 0 & \text{si} \quad (x,x) \notin \text{Arrêt} \end{cases}$$

Soit maintenant x_0 le code de $\mathcal{P}$ ($\#\mathcal{P} = x_0$). Par définition du prédicat Arrêt, on a, pour tout entier naturel x, :

$$\text{Arrêt}(x, x_0) \Leftrightarrow (x,x) \notin \text{Arrêt}.$$

En donnant à x la valeur x_0, on obtient :

$$\text{Arrêt}(x_0, x_0) \Leftrightarrow \neg\text{Arrêt}(x_0, x_0),$$

ce qui est manifestement contradictoire. ■

Remarque 3.4 Le théorème 3.9 fournit un exemple de fonction qui n'est pas calculable, à savoir la fonction indicatrice du prédicat Arrêt,

3.3 Les programmes universels

3.3.1 Le théorème d'universalité

Pour tout entier naturel $n \geq 1$, on définit la fonction $\Phi^{(n)} \in \mathcal{F}_{n+1}$ par

$$\Phi^{(n)}(x_1, \ldots, x_n, y) = \begin{cases} \psi_{\mathcal{P}}^{(n)}(x_1, \ldots, x_n) \text{ si } y = \#\mathcal{P} \\ 0 \text{ si } y \text{ n'est le code d'aucun programme} \end{cases}$$

Le théorème suivant est un des résultats fondamentaux de la théorie de la calculabilité.

Théorème 3.10 (Théorème d'universalité). *Pour chaque $n \geq 1$, la fonction $\Phi^{(n)}$ est calculable.*

Démonstration : La démonstration du théorème 3.10 réclamant un codage particulier des états du système au cours de la simulation de l'exécution du programme de code y, elle est rejetée en annexe (cf. le paragraphe A.3 de l'annexe), où le lecteur intéressé pourra la consulter. ■

On va se contenter de démontrer le théorème d'universalité dans le cas particulier suivant :

Corollaire 3.11 *Soit $m \geq 1$ un entier naturel fixé. Alors pour tout $n \geq 1$, il existe un programme $\mathcal{U}_n^m$ qui vérifie :*

si x_{n+1} est le code d'un programme dont toutes les variables sont parmi $Y, X_1, Z_1, \ldots, X_m, Z_m$, alors

$$\psi_{\mathcal{U}_n^m}^{(n+1)}(x_1, \ldots, x_n, x_{n+1}) = \Phi^{(n)}(x_1, \ldots, x_n, x_{n+1}).$$

Démonstration : On peut supposer sans danger que $m \geq n$ et on pose $p = 2m + 1$. Voici le programme $\mathcal{U}_n^m$:

$$
\begin{array}{ll}
 & \text{IF } \pi_1(X_{n+1}) = 0 \text{ GOTO } E_1 \\
 & Z_2 \leftarrow X_1 \\
 & Z_4 \leftarrow X_2 \\
 & \vdots \\
 & Z_{2n} \leftarrow X_n \\
 & Z_{p+1} \leftarrow \pi_1(X_{n+1}) \\
 & Z_{p+2} \leftarrow \tau_p(Z_1, \ldots, Z_p) \\
 & Z_{p+3} \leftarrow 1 \\
[D] & \text{IF } Z_{p+3} = Z_{p+1} + 1 \text{ GOTO } E_2 \\
 & Z_{p+5} \leftarrow \pi_2(X_{n+1}) \\
 & Z_{p+6} \leftarrow \varphi(Z_{p+1}, Z_{p+3}, Z_{p+5}) \\
 & Z_{p+4} \leftarrow \pi_2(Z_{p+6}) \\
 & \text{IF } \pi_1(Z_{p+4}) = 0 \text{ GOTO } A_1 \\
 & \text{IF } \varphi(p, \pi_2(Z_{p+4}) + 1, Z_{p+2}) = 0 \text{ GOTO } B_1 \\
 & \text{IF } \pi_1(Z_{p+4}) = 1 \text{ GOTO } C_1 \\
 & Z_{p+3} \leftarrow \min\limits_{i \leq Z_{p+1}} [\pi_1 \circ \varphi(Z_{p+1}, i, Z_{p+5}) + 1 = \pi_1(Z_{p+6})] \\
 & \text{GOTO } D \\
[A_1] & \text{IF } \pi_2(Z_{p+4}) + 1 = 1 \text{ GOTO } A_2 \\
 & \vdots \\
 & \text{IF } \pi_2(Z_{p+4}) + 1 = p \text{ GOTO } A_{p+1} \\
[A_2] & Z_1 \leftarrow Z_1 + 1 \\
 & \text{GOTO } B_2 \\
 & \vdots \\
[A_{p+1}] & Z_p \leftarrow Z_p + 1 \\
 & \text{GOTO } B_2 \\
[C_1] & \text{IF } \pi_2(Z_{p+4}) + 1 = 1 \text{ GOTO } C_2 \\
 & \vdots \\
 & \text{IF } \pi_2(Z_{p+4}) + 1 = p \text{ GOTO } C_{p+1} \\
[C_2] & Z_1 \leftarrow Z_1 - 1 \\
 & \text{GOTO } B_2 \\
 & \vdots \\
[C_{p+1}] & Z_p \leftarrow Z_p - 1 \\
[B_2] & Z_{p+2} \leftarrow \tau_p(Z_1, \ldots, Z_p) \\
[B_1] & Z_{p+3} \leftarrow Z_{p+3} + 1 \\
 & \text{GOTO } D \\
[E_2] & Y \leftarrow Z_1
\end{array}
$$

On va maintenant expliquer le fonctionnement du programme en le commentant morceau par morceau.

Soit $\mathcal{P}$ le programme de code x_{n+1} ($\#\mathcal{P} = x_{n+1}$). Soit ℓ la longueur de $\mathcal{P}$: si $\mathcal{P} = (I_1, \ldots, I_\ell)$, alors $x_{n+1} = \langle \ell, \tau_\ell(\#I_1, \ldots, \#I_\ell) \rangle$. Pour chaque $i \in \{1, \ldots, \ell\}$, $\#I_i = \langle a_i, \langle b_i, c_i \rangle \rangle$.

On code l'état courant de $\mathcal{P}$ par $\tau_p(y, x_1, z_1, \ldots, x_m, z_m)$, si $y, x_1, z_1, \ldots, x_m, z_m$ désignent les valeurs respectives des variables $Y, X_1, Z_1, \ldots, X_m, Z_m$ de $\mathcal{P}$.

On commence par regarder si $\mathcal{P}$ est le programme vide, auquel cas on s'arrête tout de suite. Si la valeur initiale de X_{n+1} n'est le code d'aucun programme, **par convention**, on s'arrête tout de suite également [on rappelle que les programmes non vides sont codés par tous les nombres y tels que $\pi_1(y) \neq 0$.] D'où l'instruction

$$\text{IF } \pi_1(X_{n+1}) = 0 \text{ GOTO } E_1$$

On simule la variable de sortie de $\mathcal{P}$ par la variable Z_1 de $\mathcal{U}_n^m$, les variables Z_i ($1 \leq i \leq m$) de $\mathcal{P}$ par les variables Z_{2i+1} de $\mathcal{U}_n^m$ et les variables X_i (dont les n premières sont en quelque sorte communes à $\mathcal{P}$ et $\mathcal{U}_n^m$!) par les variables Z_{2i} de $\mathcal{U}_n^m$. D'où l'initialisation de ces dernières puis l'affectation à la variable Z_{p+2} du code de l'état initial

$$\begin{array}{l} Z_2 \leftarrow X_1 \\ Z_4 \leftarrow X_2 \\ \vdots \\ Z_{2n} \leftarrow X_n \\ Z_{p+2} \leftarrow \tau_p(Z_1, \ldots, Z_p) \end{array}$$

On donne ensuite à Z_{p+1} la valeur de la longueur de $\mathcal{P}$ et on se met en position pour simuler la 1ère instruction de $\mathcal{P}$ (la valeur de la variable Z_{p+3} sera le numéro d'ordre de l'instruction de $\mathcal{P}$ qu'on est sur le point de simuler) :

$$\begin{array}{l} Z_{p+1} \leftarrow \pi_1(X_{n+1}) \\ Z_{p+3} \leftarrow 1 \end{array}$$

Si l'instruction à simuler est la $(\ell+1)$ème de $\mathcal{P}$, alors la simulation de l'exécution de $\mathcal{P}$ est terminée et on se rend en F (où la variable de sortie de $\mathcal{U}_n^m$ recevra la valeur convenable) :

$$[D] \quad \text{IF } Z_{p+3} = Z_{p+1} + 1 \text{ GOTO } E_2$$

Si la simulation de l'exécution de $\mathcal{P}$ n'est pas terminée ($1 \leq Z_{p+3} \leq \ell$), il faut décoder l'instruction à simuler. Z_{p+5} reçoit la valeur $\tau_\ell(\#I_1, \ldots, \#I_\ell)$, Z_{p+6} reçoit la valeur $\#I_{z_{p+3}}$ et Z_{p+4} reçoit la valeur $\langle b_{z_{p+3}}, c_{z_{p+3}} \rangle$, c'est-à-dire le code de **l'action** à simuler :

$$\begin{aligned}
&Z_{p+5} \leftarrow \pi_2(X_{n+1})\\
&Z_{p+6} \leftarrow \varphi(Z_{p+1}, Z_{p+3}, Z_{p+5})\\
&Z_{p+4} \leftarrow \pi_2(Z_{p+6})
\end{aligned}$$

Si $b_{z_{p+3}} = 0$, l'action à simuler est une incrémentation et on se rend en A_1 (où on simulera l'incrémentation) :

$$\text{IF } \pi_1(Z_{p+4}) = 0 \text{ GOTO } A_1$$

Si $b_{z_{p+3}} \neq 0$, l'action à simuler est une décrémentation $V \leftarrow V - 1$ ou un branchement conditionelle IF $V \neq 0$ GOTO L ; dans les deux cas, si la variable V (dont le code $\#V$ est donné par $\pi_2(Z_{p+4}) + 1$) vaut 0, l'état courant ne doit pas être modifié et on passe à la simulation de l'instruction de $\mathcal{P}$ qui suit immédiatement celle qu'on est en train de simuler, ce pour quoi on se rend immédiatement en B_1 (où on se content d'incrémenter Z_{p+3}, puis de retourner en D) :

$$\text{IF } \varphi(p, \pi_2(Z_{p+4}) + 1, Z_{p+2}) = 0 \text{ GOTO } B_1$$

Si $b_{z_{p+3}} = 1$ et la variable V ne vaut pas 0, il faut décrémenter V, ce pour quoi on se rend en C_1 :

$$\text{IF } \pi_1(Z_{p+4}) = 1 \text{ GOTO } C_1$$

Si $b_{z_{p+3}} > 1$ et la variable V ne vaut pas 0, il faut poursuivre en simulant l'instruction de $\mathcal{P}$ qui porte l'étiquette de code $b_{z_{p+3}} - 1$; pour cela, on repère dans $\mathcal{P}$ la première instruction I_i dont l'étiquette est celle qu'on cherche [On remarque que si aucune instruction de $\mathcal{P}$ ne porte l'étiquette recherchée, alors Z_{p+3} prend la valeur $\ell + 1$, ce qui provoquera l'arrêt de l'exécution de $\mathcal{P}$ grâce à l'instruction d'étiquette D de $\mathcal{U}_n^m$] :

$$\begin{aligned}
&Z_{p+3} \leftarrow \min_{i \leq Z_{p+1}} [\pi_1 \circ \varphi(Z_{p+1}, i, Z_{p+5}) + 1 = \pi_1(Z_{p+6})]\\
&\text{GOTO } D_1
\end{aligned}$$

Quand on doit simuler une incrémentation, on se reporte à la variable à incrémenter, puis on calcule le code du nouvel état courant, avant de poursuivre par la simulation de l'instruction suivante de $\mathcal{P}$:

$$\begin{array}{ll}
[A_1] & \text{IF } \pi_2(Z_{p+4})+1=1 \text{ GOTO } A_2 \\
 & \vdots \\
 & \text{IF } \pi_2(Z_{p+4})+1=p \text{ GOTO } A_{p+1} \\
[A_2] & Z_1 \leftarrow Z_1+1 \\
 & \text{GOTO } B_2 \\
 & \vdots \\
[A_{p+1}] & Z_p \leftarrow Z_p+1 \\
 & \text{GOTO } B_2 \\
 & \vdots \\
[B_2] & Z_{p+2} \leftarrow \tau_p(Z_1,\ldots,Z_p) \\
[B_1] & Z_{p+3} \leftarrow Z_{p+3}+1 \\
 & \text{GOTO } D
\end{array}$$

Quand on doit simuler une décrémentation, on se reporte à la variable à décrémenter, puis on calcule le code du nouvel état courant, avant de poursuivre par la simulation de l'instruction suivante de $\mathcal{P}$:

$$\begin{array}{ll}
[C_1] & \text{IF } \pi_2(Z_{p+4})+1=1 \text{ GOTO } C_2 \\
 & \vdots \\
 & \text{IF } \pi_2(Z_{p+4})+1=p \text{ GOTO } C_{p+1} \\
[C_2] & Z_1 \leftarrow Z_1-1 \\
 & \text{GOTO } B_2 \\
 & \vdots \\
[C_{p+1}] & Z_p \leftarrow Z_p-1 \\
[B_2] & Z_{p+2} \leftarrow \tau_p(Z_1,\ldots,Z_p) \\
[B_1] & Z_{p+3} \leftarrow Z_{p+3}+1 \\
 & \text{GOTO } D
\end{array}$$

Enfin, si la simulation de $\mathcal{P}$ se termine, on donne à Y la valeur de sortie de $\mathcal{P}$:

$$[E_2] \quad Y \leftarrow Z_1$$

■

3.3.2 Récursivité et calculabilité

Dans cette section, on donne une suite de résultats (repris et démontrés dans l'annexe A qui se trouve à la fin de ces notes de cours) qui conduisent à l'équivalence entre la notion de "fonction partielle récursive" et celle de "fonction partielle calculable".

Soit n un entier naturel ≥ 1. On définit le prédicat $(n+2)$-aire $\mathrm{STP}^{(n)}$ de la façon suivante :

$$\mathrm{STP}^{(n)}(x_1, \ldots, x_n, y, t) \quad \Leftrightarrow \quad \begin{cases} y \text{ est le code d'un programme dont le calcul,} \\ \text{sur les valeurs d'entrées initiales } x_1, \ldots, x_n, \\ \text{est de longueur } k \leq t+1. \end{cases}$$

Une petite modification du programme $\mathcal{U}_n$ dans la démonstration du théorème 3.10 (cf. le paragraphe A.3 de l'annexe) conduirait au résultat suivant :

Corollaire 3.12 *Pour tout entier $n \geq 1$, le prédicat $STP^{(n)}$ est calculable.* ■

On peut faire un peu mieux :

Théorème 3.13 *Pour tout entier naturel $n \geq 1$, le prédicat $STP^{(n)}$ est primitif récursif.*

Démonstration : cf. le paragraphe A.4 de l'annexe. ■

En utilisant une méthode voisine, on montre aussi le résultat suivant :

Théorème 3.14 (Théorème de la forme normale) *Soit $f \in \mathcal{F}_n$ une fonction partielle calculable. Alors il existe un prédicat primitif récursif R, d'arité $n+1$, qui satisfait*

$$f(x_1, \ldots, x_n) = \pi_1(\min_z[R(x_1, \ldots, x_n, z)]).$$

Démonstration : cf. le paragraphe A.5 de l'annexe. ■

Il est alors facile de montrer le théorème annoncé dans le chapitre 2 :

Théorème 3.15 *L'ensemble des fonctions partielles calculables est égal à l'ensemble des fonctions partielles récursives.*

Démonstration : Il suffit de montrer que toute fonction partielle calculable est récursive, l'autre inclusion ayant été démontrée dans le chapitre 2 (cf. le théorème 2.15). Soit f une fonction partielle calculable de $\mathcal{F}_n$. D'après le théorème de la forme normale, f est égale à $\pi_1 \circ g$, où g est obtenue par minimisation (non bornée) à partir d'un prédicat primitif récursif. f est donc récursive. ■

3.4 Énumérations effectives

Dans cette section, on s'intéresse aux fonctions calculables de $\mathbb{N}$ dans $\mathbb{N}$.

Définition 3.3 *Soit $\mathcal{H}$ un sous-ensemble de $\mathcal{F}_1$ et f une fonction récursive $f \in \mathcal{F}_2$. Pour tout $y \in \mathbb{N}$, on pose $f_y(x) \stackrel{déf}{=} f(x, y)$. On dit que $\mathcal{H}$ est* **récursivement énuméré**[2] **par** *f si*

$$\mathcal{H} = \{f_y \mid y \in \mathbb{N}\}$$

La définition 3.3 nous permet de reformuler le théorème d'universalité pour les fonctions unaires.

Théorème 3.16 *L'ensemble des fonctions (partielles) récursives de $\mathcal{F}_1$ est récursivement énuméré par $\Phi^{(1)}$.* ■

Théorème 3.17 *Il n'existe pas de fonction récursive $g \in \mathcal{F}_2$ telle l'ensemble des fonctions totales récursives de $\mathcal{F}_1$ soit récursivement énuméré par g.*

Démonstration : Supposons le contraire : soit alors $g \in \mathcal{F}_2$ une fonction récursive qui énumère les fonctions totales récursives de $\mathcal{F}_1$. Soit alors la fonction $h \in \mathcal{F}_1$ définie par

$$h(x) = g(x, x) + 1.$$

C'est une fonction totale (car pour tout x, g_x est une fonction totale !) et récursive (car g est récursive !) Il existe donc un entier naturel y tel que $h = g_y$. On a alors

$$h(y) = g(y, y) = g(y, y) + 1,$$

d'où la contradiction. ■

Les théorèmes 3.16 et 3.17 permettent de mieux comprendre la démarche suivie dans le chapitre 2 : les fonctions primitives récursives ne suffisant pas à "attraper" toutes les fonctions totales calculables (cf. la fonction d'Ackermann), il a fallu passer à un ensemble plus grand ; malheureusement, les fonctions totales récursives ne conviennent pas, car elles ne peuvent être récursivement énumérées, d'où le passage aux fonctions partielles récursives !

[2]On dit aussi **effectivement énuméré**.

3.5 Le théorème de Rice

On manque jusqu'à présent de techniques permettant de montrer qu'un ensemble donné d'entiers naturels n'est pas récursif (on dit alors que cet ensemble est **indécidable**). Le théorème permet de combler partiellemnt cette lacune. Après avoir fixé la notation et établi un nouveau résultat d'indécidabilité, on énoncera — sans le démontrer — un lemme d'aspect assez surprenant, qui nous livrera le théorème attendu.

Notation. Pour tout $i \in \mathbb{N}$, on pose $\varphi_i(x) \stackrel{déf}{=} \Phi^{(1)}(x, i)$; autrement dit, φ_i est la fonction de $\mathcal{F}_1$ calculée par le programme de code i. Attention ! Il va de soi que pour toute fonction calculable $f \in \mathcal{F}_1$, il existe une infinité d'entiers naturels i tels que $f = \varphi_i$. Un tel i est appelé un indice de f.

On rappelle que pour toute fonction $f \in \mathcal{F}_p$ et tout $\overline{x} \in \mathbb{N}^p$, l'expression $f(\overline{x})\downarrow$ signifie que $\overline{x}$ appartient à l'ensemble de définition de f. On pose

$$K \stackrel{déf}{=} \{n \mid \varphi_n(n)\downarrow\}.$$

Proposition 3.18 *L'ensemble K n'est pas récursif.*

Démonstration : Il s'agit d'un corollaire évident de l'indécidabilité du problème de l'arrêt. En effet,

$$\begin{array}{rcl} n \in K & \Leftrightarrow & \Phi^{(1)}(n, n)\downarrow \\ & \Leftrightarrow & \text{Arrêt}(n, n) \end{array}$$

La démonstration du théorème 3.9 (indécidabilité du problème de l'arrêt) montre alors que supposer K récursif mène à une contradiction. ∎

Lemme 3.19 *Soit $\epsilon : \mathbb{N} \to \mathbb{N}$ la fonction partielle vide de $\mathcal{F}_1$ (ϵ n'est définie pour aucune entier). Soit $\varphi : \mathbb{N} \to \mathbb{N}$ une fonction partielle calculable de domaine non vide. Alors il existe une fonction totale calculable $f : \mathbb{N} \to \mathbb{N}$ telle que*

$$\varphi_{f(i)} = \begin{cases} \varphi & si\ i \in K \\ \epsilon & sinon \end{cases} \quad ∎$$

Théorème 3.20 (Théorème de Rice) *Soit I un sous-ensemble propre de $\mathbb{N}$ (i.e. $I \neq \emptyset$ et $I \neq \mathbb{N}$). Si I est récursif, alors il existe i, j tels $i \in I$, $j \in \mathbb{N} - I$ et $\varphi_i = \varphi_j$.*

Démonstration : Supposons le contraire (*i.e.* $\varphi_i \neq \varphi_j$ dès que $i \in I$ et $j \in \mathbb{N} - I$). Alors, en échangeant au besoin les rôles des I et $\mathbb{N} - I$, on peut supposer que I contient l'indice (et par conséquent tous les indices) d'une fonction φ non vide et que $\mathbb{N} - I$ contient un indice (et par conséquent tous les indices) de la fonction vide ϵ. Soit $f : \mathbb{N} \to \mathbb{N}$ la fonction totale calculable définie comme dans le lemme 3.19 ci-dessus. Alors $\mathbf{1}_I \circ f : \mathbb{N} \to \{0, 1\}$ est une fonction totale calculable. Si $\varphi_{f(i)} = \epsilon$, alors $f(i)$, étant un indice de ϵ, appartient à $\mathbb{N} - I$; partant, si $f(i) \in I$, alors $\varphi_{f(i)} \neq \epsilon$, d'où $\varphi_{f(i)} = \varphi$. D'autre part, si $\varphi_{f(i)} = \varphi$, alors $f(i)$, étant un indice de φ, appartient à I. Finalement,

$$\begin{aligned} \mathbf{1}_I(f(i)) = 1 \quad &\Leftrightarrow \quad \varphi_{f(i)} = \varphi \\ &\Leftrightarrow \quad i \in K \end{aligned}$$

Puisque $\mathbf{1}_I \circ f$ est calculable, K est récursif, ce qui contredit la proposition 3.18. ■

Définition 3.4 *On dit qu'un sous-ensemble I de $\mathbb{N}$* **respecte les indices** *si $\varphi_i \neq \varphi_j$ dès que $i \in I$ et $j \in \mathbb{N} - I$.*

On obtient ainsi la reformulation suivante du théorème de Rice.

Corollaire 3.21 *Un sous-ensemble propre de $\mathbb{N}$ qui respecte les indices n'est pas récursif.* ■

À titre d'illustration de l'utilité de ce corollaire, on peut montrer qu'aucun des sous-ensembles suivants de $\mathbb{N}$ n'est récursif :

$$\begin{array}{l} \{i \in \mathbb{N} \mid \varphi_i = \varphi_j\} \text{ où } j \text{ est un entier naturel fixé} \\ \{i \in \mathbb{N} \mid \varphi_i \text{ est totale}\} \\ \{i \in \mathbb{N} \mid \varphi_i(a) \downarrow\} \text{ où } a \text{ est un entier naturel fixé} \\ \{i \in \mathbb{N} \mid \varphi_i \text{ est constante}\} \\ \{i \in \mathbb{N} \mid \text{domaine}(\varphi_i) \text{ est fini}\} \end{array}$$

Annexe A

Compléments sur l'universalité

A.1 Rappels arithmétiques

La fonction $g \in \mathcal{F}_2$ définie par

$$g(x_1, x_2) = \min_{t \leq x_2}[\text{Premier}(t) \wedge t > x_1]$$

est primitive récursive, car obtenue par minimisation bornée à partir d'un prédicat primitif récursif.

La fonction $h \in \mathcal{F}_1$ définie par

$$h(x) = g(x, x! + 1)$$

est primitive récursive, car c'est la composée de deux fonctions primitives récursives.

Le résultat suivant est un grand classique sur la répartition des nombres premiers :

Proposition A.1 *Soit p un nombre premier. Alors il existe un nombre premier q tel que*

$$p < q \leq p! + 1.$$

On montre alors que la fonction pr de $\mathcal{F}_1$ définie par

$$\text{pr}(x) = \begin{cases} 0 \text{ si } x = 0 \\ \text{le } x\text{ème nombre premier si } x \geq 1 \end{cases}$$

est primitive récursive car

$$\begin{cases} \text{pr}(0) & = & 0 \\ \text{pr}(t+1) & = & h(\text{pr}(t)) \end{cases}$$

En effet, d'après la proposition A.1 ci-dessus, pour tout entier naturel n, on a

$$\mathrm{pr}(n) < \mathrm{pr}(n+1) \leq \mathrm{pr}(n)! + 1,$$

A.2 Codage des états

Le symbole V_i ($i \geq 1$) désigne la variable occupant la position i dans l'énumération[1] du paragraphe 3.1.3 (V_1 désigne la variable de sortie Y, V_{2i} désigne la variable d'entrée X_i et V_{2i+1} désigne la variable auxiliaire Z_i.)

Soit σ un état dans lequel V_i a pour valeur v_i. On rappelle que dans un état, toutes les variables à l'exception d'un nombre fini ont une valeur nulle. Comme $p^0 = 1$, le produit infini

$$\prod_{i \geq 1} \mathrm{pr}(i)^{v_i}$$

a une valeur finie qu'on appelle le **code de Gödel de** σ et qu'on note $\mathrm{Göd}(\sigma)$.

Proposition A.2 *La fonction qui à σ associe son code de Gödel est une bijection de l'ensemble des états dans l'ensemble des entiers naturels non nuls.* ■

En utilisant l'unicité de la décomposition en facteurs premiers d'un nombre entier ≥ 1, on montre facilement :

Proposition A.3 *Soit σ un état dont le code de Gödel est s.*

- *La valeur de V_i dans l'état σ est non nulle si et seulement si $pr(i)$ divise s ;*
- *La valeur de la variable Y dans l'état σ est égale au plus grand entier z tel que 2^z divise s ;*
- *Le code de Gödel de l'état σ' obtenu à partir de σ en ajoutant 1 à la valeur de V_i est $s \cdot pr(i)$;*
- *Si V_i est une variable dont la valeur dans l'état σ est non nulle, alors le code de Gödel de l'état σ' obtenu à partir de σ en ôtant 1 à la valeur de V_i est s **div** $pr(i)$.* ■

On montre aussi que la fonction β_i de $\mathcal{F}_1$ ($i \geq 1$) qui retourne le plus grand entier z tel que $(\mathrm{pr}(i))^z$ divise s est primitive récursive :

$$\beta_i(s) = \kappa_i(s,s) \quad \text{où} \quad \kappa_i(s,y) = \min_{z \leq y}[\mathrm{Divise}((\mathrm{pr}(i))^z, s) \wedge \neg \mathrm{Divise}((\mathrm{pr}(i))^{z+1}, s)].$$

[1] On rappelle qu'on compte à partir de 1 !

Partant, la fonction $\rho \in \mathcal{F}_2$ définie par

$$\rho(i,s) = \begin{cases} 0 & \text{si } i = 0 \\ \beta_i(s) & \text{sinon} \end{cases}$$

est primitive récursive.

A.3 Le théorème d'universalité

Pour tout entier naturel $n \geq 1$, on définit la fonction $\Phi^{(n)} \in \mathcal{F}_{n+1}$ par

$$\Phi^{(n)}(x_1, \ldots, x_n, y) = \begin{cases} \psi_{\mathcal{P}}^{(n)}(x_1, \ldots, x_n) \text{ si } y = \#\mathcal{P} \\ 0 \text{ si } y \text{ n'est le code d'aucun programme} \end{cases}$$

Théorème 3.10 (Théorème d'universalité) *Pour tout $n \geq 1$, la fonction $\Phi^{(n)}$ est calculable.*

Démonstration : Voici un programme $\mathcal{U}_n$ qui calcule $\Phi^{(n)}$:

$$\begin{array}{ll}
 & \text{IF } \pi_1(X_{n+1}) = 0 \text{ GOTO } E_1 \\
 & Z_{p+1} \leftarrow \pi_1(X_{n+1}) \\
 & Z_{p+2} \leftarrow \prod_{i=1}^{n} (\text{pr}(2i))^{X_i} \\
 & Z_{p+3} \leftarrow 1 \\
[D] & \text{IF } Z_{p+3} = Z_{p+1} + 1 \text{ GOTO } E_2 \\
 & Z_{p+5} \leftarrow \pi_2(X_{n+1}) \\
 & Z_{p+6} \leftarrow \varphi(Z_{p+1}, Z_{p+3}, Z_{p+5}) \\
 & Z_{p+4} \leftarrow \pi_2(Z_{p+6}) \\
 & \text{IF } \pi_1(Z_{p+4}) = 0 \text{ GOTO } A \\
 & \text{IF } \neg\text{Divise}(\text{pr}(\pi_2(Z_{p+4}) + 1), Z_{p+2}) \text{ GOTO } B \\
 & \text{IF } \pi_1(Z_{p+4}) = 1 \text{ GOTO } C \\
 & Z_{p+3} \leftarrow \min_{i \leq Z_{p+1}} [\pi_1 \circ \varphi(Z_{p+1}, i, Z_{p+5}) + 1 = \pi_1(Z_{p+6})] \\
 & \text{GOTO } D \\
[A] & Z_{p+2} \leftarrow Z_{p+2} \cdot \text{pr}(\pi_2(Z_{p+4}) + 1) \\
 & \text{GOTO } B \\
[C] & Z_{p+2} \leftarrow Z_{p+2} \,\mathbf{div}\, \text{pr}(\pi_2(Z_{p+4}) + 1) \\
[B] & Z_{p+3} \leftarrow Z_{p+3} + 1 \\
 & \text{GOTO } D \\
[E_2] & Y \leftarrow \rho(1, Z_{p+2})
\end{array}$$

■

A.4 Le théorème de comptage des étapes

Soit n un entier naturel ≥ 1. On définit le prédicat $(n+2)$-aire $\mathrm{STP}^{(n)}$ de la façon suivante :

$$\mathrm{STP}^{(n)}(x_1, \ldots, x_n, y, t) \quad \Leftrightarrow \quad \begin{cases} y \text{ est le code d'un programme dont le calcul,} \\ \text{sur les valeurs d'entrées initiales } x_1, \ldots, x_n, \\ \text{est de longueur } k \leq t+1. \end{cases}$$

Théorème 3.13 *Pour tout entier naturel $n \geq 1$, le prédicat $STP^{(n)}$ est primitif récursif.*

Démonstration : On code la description instantanée (i, σ) par $\langle i, \text{Göd}(\sigma)\rangle$.

On commence par définir quelques fonctions primitives récursives qui permettent respectivement d'extraire le numéro d'ordre de l'étiquette, le numéro d'ordre de la variable, la forme de l'action et le numéro d'ordre de "l'étiquette de branchement" de la $i$ème instruction du programme de code y :

$$\begin{array}{rcl} \text{Ét}(i,y) & = & \pi_1 \circ \varphi(\pi_1(y), i, \pi_2(y)) \\ \text{Var}(i,y) & = & \pi_2 \circ \pi_2 \circ \varphi(\pi_1(y), i, \pi_2(y)) + 1 \\ \text{Act}(i,y) & = & \pi_1 \circ \pi_2 \circ \varphi(\pi_1(y), i, \pi_2(y)) \\ \text{ÉtBr}(i,y) & = & \pi_1 \circ \pi_2 \circ \varphi(\pi_1(y), i, \pi_2(y)) \dot{-} 1 \end{array}$$

On définit maintenant des prédicats primitifs récursifs qui disent quel type d'action effectuer quand on est dans la description instantanée de code x du programme de code y, à savoir passer (i.e. pas d'action à effectuer), incrémenter, décrémenter ou se brancher[2] :

$$\begin{array}{rcl} \text{Passe}(x,y) & \Leftrightarrow & \text{Act}(\pi_1(x), y) \geq 1 \wedge \neg \text{Divide}(\text{pr} \circ \text{Var}(\pi_1(x), y), \pi_2(x)) \\ \text{Incr}(x,y) & \Leftrightarrow & \text{Act}(\pi_1(x), y) = 0 \\ \text{Décr}(x,y) & \Leftrightarrow & \text{Act}(\pi_1(x), y) = 1 \wedge \text{Divide}(\text{pr} \circ \text{Var}(\pi_1(x), y), \pi_2(x)) \\ \text{Branch}(x,y) & \Leftrightarrow & \text{Act}(\pi_1(x), y) > 1 \wedge \text{Divide}(\text{pr} \circ \text{Var}(\pi_1(x), y), \pi_2(x)) \\ & & \wedge\ \exists_{i \leq \pi_1(y)} \text{Ét}(i,y) = \text{ÉtBr}(\pi_1(x), y) \end{array}$$

On est maintenant en mesure de définir la fonction primitive récursive $\text{Succ} \in \mathcal{F}_2$ telle que $\text{Succ}(x,y)$ est le code de la description instantanée (du programme de code y) qui est le successeur de la description instantanée de

[2] On rappelle qu'une quantification existentielle bornée revient à une somme bornée.

code x :

$$\mathrm{Succ}(x,y) = \begin{cases} \langle \pi_1(x)+1, \pi_2(x)\rangle & \text{si Passe}(x,y) \\ \langle \pi_1(x)+1, \pi_2(x)\cdot \mathrm{pr}(\mathrm{Var}(\pi_1(x),y))\rangle & \text{si Incr}(x,y) \\ \langle \pi_1(x)+1, \pi_2(x)\,\mathbf{div}\,\mathrm{pr}(\mathrm{Var}(\pi_1(x),y))\rangle & \text{si Décr}(x,y) \\ \langle \min\limits_{i\leq \pi_1(y)} [\text{Ét}(i,y) = \text{ÉtBr}(\pi_1(x),y)], \pi_2(x)\rangle & \text{si Branch}(x,y) \\ \langle \pi_1(y)+1, \pi_2(x)\rangle & \text{sinon} \end{cases}$$

Naturellement, on code la description instantanée initiale de la façon suivante :

$$\mathrm{Init}^{(n)}(x_1,\ldots,x_n) = \langle 1, \prod_{i=1}^{n} (\mathrm{pr}(2i))^{x_i}\rangle,$$

et on teste si la description instantanée de code x est une description instantanée terminale du programme de code y grâce au prédicat primitif récursif

$$\mathrm{Term}(x,y) \Leftrightarrow \pi_1(x) > \pi_1(y).$$

On peut maintenant définir une fonction primitive récursive $\mathrm{Inst}^{(n)} \in \mathcal{F}_{n+2}$ qui "produit" les codes des descriptions instantanées successives du calcul du programme de code y sur les variables d'entrées $x_1,\ldots,x_n$:

$$\begin{cases} \mathrm{Inst}^{(n)}(x_1,\ldots,x_n,y,0) & = \mathrm{Init}^{(n)}(x_1,\ldots,x_n) \\ \mathrm{Inst}^{(n)}(x_1,\ldots,x_n,y,t+1) & = \mathrm{Succ}(\mathrm{Inst}^{(n)}(x_1,\ldots,x_n,y,t),y) \end{cases}$$

Il ne reste plus qu'à remarquer que

$$\mathrm{STP}^{(n)}(x_1,\ldots,x_n,y,t) \Leftrightarrow \mathrm{Term}(\mathrm{Inst}^{(n)}(x_1,\ldots,x_n,y,t),y),$$

ce qui achève de prouver que $\mathrm{STP}^{(n)}$ est primitif récursif. ■

A.5 Le théorème de la forme normale

Théorème 3.14 (Théorème de la forme normale) *Soit $f \in \mathcal{F}_n$ une fonction calculable. Alors il existe un prédicat primitif récursif R, d'arité $n+1$, qui satisfait*

$$f(x_1,\ldots,x_n) = \pi_1(\min_z [R(x_1,\ldots,x_n,z)]).$$

Démonstration : Soit y_0 le code d'un programme calculant f. On définit le prédicat R, d'arité $n+1$, par :

$$R(x_1,\ldots,x_n,z) \Leftrightarrow \left\{ \begin{array}{l} \mathrm{STP}^{(n)}(x_1,\ldots,x_n,y_0,\pi_2(z)) \\ \wedge\ \rho(1,\pi_2(\mathrm{Inst}^{(n)}(x_1,\ldots,x_n,y_0,\pi_2(z)))) = \pi_1(z) \end{array} \right.$$

R est primitif récursif. Il nous suffit donc de montrer que

$$f(x_1,\ldots,x_n) = \pi_1(\min_z[R(x_1,\ldots,x_n,z)]).$$

<u>1er cas</u> : $\pi_1(\min_z[R(x_1,\ldots,x_n,z)])$ est défini. Alors il existe un z tel que

$$\begin{array}{l} \mathrm{STP}^{(n)}(x_1,\ldots,x_n,y_0,\pi_2(z)) \\ \wedge\ \rho(1,\pi_2(\mathrm{Inst}^{(n)}(x_1,\ldots,x_n,y_0,\pi_2(z)))) = \pi_1(z) \end{array}$$

Pour un tel z, le calcul du programme de code y sur les valeurs d'entrées $x_1,\ldots,x_n$ atteint en au plus $\pi_2(z)$ étapes une description instantanée terminale dans laquelle la variable de sortie Y a la valeur $\pi_1(z)$, c'est-à-dire $\pi_1(z) = f(x_1,\ldots,x_n)$.

<u>2ème cas</u> : $\pi_1(\min_z[R(x_1,\ldots,x_n,z)])$ n'est pas défini. Alors pour tout t,

$$\neg\mathrm{STP}^{(n)}(x_1,\ldots,x_n,y_0,t),$$

c'est-à-dire $f(x_1,\ldots,x_n) =\uparrow$. ■

Bibliographie

[Davis *et alii*] Martin DAVIS, Ron SIGAL et Elaine WEYUKER, *Computability, Complexity and Languages*, 2ème édition, San Diego, Academic Press, 1994.

[Cori-Lascar] René CORI et Daniel LASCAR, *Logique mathématique*, 2 volumes, Paris, Masson, 1993.

[Smith] Carl H. SMITH, *A Recursive Introduction to the Theory of Computation*, Graduate Texts in Computer Science, New York, Springer-Verlag, 1994.

[Stern] Jacques STERN, *Fondements mathématiques de l'informatique*, McGraw-Hill, 1990.