Introduction à la calculabilité

Benoit Monin

17 septembre 2018

Table des matières

Chapitre 1

Premières intuitions

1.1 Notations

On note $\mathbb{N}$ l'ensemble des entiers naturels. On note $\mathbb{N}^*$ l'ensemble des entiers naturels strictement positifs. Pour deux ensemble A, B, on note A^B l'ensemble des fonctions de B dans A. Pour deux ensembles A, B, on note $A \times B$ l'ensemble des couples (a, b) pour $a \in A$ et $b \in B$. Notez que les couples sont ordonnés : le couple (a, b) est différent du couple (b, a). Soit A un ensemble et un entier $n > 1$. On notera A^n l'ensemble $A \times A^{n-1}$ (par exemple $\mathbb{N}^3$ est égale à $\mathbb{N} \times \mathbb{N} \times \mathbb{N}$, l'ensemble de tous les triplets (n_1, n_2, n_3) tels que $n_1, n_2, n_3 \in \mathbb{N}$).

On rapelle les définitions suivantes : soit deux ensembles A et B. Soit une fonction $f : A \to B$. On dit que f est une injection si deux éléments de A distincts sont toujours envoyés vers deux éléments de B distincts : $\forall a_1 \neq a_2 \in A$ on a $f(a_1) \neq f(a_2)$. On dit que f est une surjection si l'image de A par f recourve tout l'ensemble B : $\forall b \in B \ \exists a \in A \ f(a) = b$. On dit que f est une bijection si f est à la fois une injection et une surjection.

On travaillera aussi avec des fonctions partiels, c'est à dire qui ne sont pas forcément définies partout. Pour une fonction partielle $f : \mathbb{N}^n \to \mathbb{N}$, on écriera $f(x_1, \ldots, x_n) \uparrow$ si la fonction n'est pas définie sur les valeurs $x_1, \ldots, x_n$. On écriera $f(x_1, \ldots, x_n) \downarrow$ dans le cas contraire, et $f(x_1, \ldots, x_n) \downarrow = y$ pour spécifier que la fonction est égale à y sur les valeurs $x_1, \ldots, x_n$.

1.2 Intuitions

1.2.1 Les choses calculables

Nous allons conduire une études théorique de la puissance de calcul des programmes informatiques. Nous nous restreignons pour cela à la calculabilité d'objets mathématiques simples, comme par exemple les nombres réels, les sous-ensembles de $\mathbb{N}$, ou encore les fonctions de $\mathbb{N}$ dans $\mathbb{N}$. Intuitivement, un nombre réel R est calculable, si il existe un programme informatique qui peut écrire dans l'ordre les décimales de R sans jamais se tromper. Évidemment si le nombre réel est irrationnel, comme par exemple $\sqrt{2}, \pi$ ou encore le nombre d'or, il y a une infinité de décimales qui ne se répètent jamais. Aussi un programme informatique qui s'exécute n'aura jamais terminé d'écrire le nombre réel en entier. Ce n'est toutefois pas ce qui est demandé : pour qu'un nombre réel R soit calculable, il faut qu'il existe un programme qui puisse écrire les décimales de R sans jamais se tromper, de telle manière que pour n'importe quel entier n, si l'on attend suffisamment longtemps, le programme aura écrit les n premières décimales de R. On partira également du principe que notre programme s'exécute sur une machine qui peut utiliser autant de mémoire qu'elle le souhaite. Ce n'est évidemment pas le cas en pratique : la mémoire

des ordinateurs est bornée. Il apparaît toutefois nécessaire pour notre étude théorique, de s'affranchir de cette limitation malheureuse. Essayons de justifier cela par un exemple.

Soit le réel R dont l'écriture en base 2 contient un 1 en position i si et seulement si i est un nombre premier. Moralement ce réel devrait être calculable : il existe un programme informatique simple qui permet de déterminer si un entier naturel est premier ou non. Il suffit donc de faire une boucle sur tous les entiers, puis d'écrire 0 si l'entier numéro i n'est pas premier, et 1 sinon. En pratique, quand les entiers naturels sont trop grands pour être représentés sur la mémoire d'une machine, il n'y a pas grand chose que nous puissions faire. En ce qui nous concerne, ces limitations ne nous affectent pas, et nous raisonnerons de manière purement théorique, en imaginant une machine "idéale", pouvant utiliser une mémoire non bornée.

1.2.2 Les choses incalculables

La première question que l'on peut se poser est : existe-t-il des choses que l'on ne peut pas calculer ? Attention, on ne tient pas compte ici du temps de calcul nécessaire. Il y a beaucoup de choses que l'on sait calculer en théorie, mais qui en pratique prendrait trop de temps. Un exemple classique est le problème du voyageur de commerce : étant donné 100 villes, quel est le plus cours chemin qui relie toutes les villes entre elles une et une seule fois ? Il est très simple d'écrire un programme informatique qui calcule la solution à ce problème. Essentiellement : "Pour tous les chemins possibles, calculer la taille du chemin, et retenir le plus court". Toutefois le nombres de chemins possibles pour 100 villes étant bien supérieur au nombre estimé d'atomes dans l'univers, on voit mal comment même l'ensemble de tous les ordinateurs du monde puissent terminer le calcul de cet algorithme en un temps qui soit à échelle humaine.

Encore une fois, notre étude est purement théorique, et de notre point de vue, il n'y a pas de différence entre le problème du voyageur de commerce et la fonction de multiplication par 2 : ces deux choses sont calculables et on ne fera pas d'autres distinctions. Existe-t-il alors des nombres qui ne sont pas calculables ? On ne parle pas ici de nombres qu'on ne peut pas "encore" calculer, et que l'on pourra peut-être calculer un jour quand on aura des ordinateurs plus puissants. On parle bien ici de nombres qui sont *fondamentalement incalculables*. Des nombres que l'on ne pourra jamais calculer, ni aujourd'hui, ni demain, et ce indépendamment de quelques progrès techniques que ce soit.

La réponse est oui. Nous faisons appel pour cela à un théorème fondamental en logique mathématique, qui fut décliné par la suite de très nombreuses manières différentes.

Définition 1.2.1 : Un ensemble infini A est dit dénombrable si il existe une bijection $f : \mathbb{N} \to A$. Un ensemble infini A est dit indénombrable si il n'existe pas de bijection $f : \mathbb{N} \to A$. ◊

Exercice 1.2.2

Soit A un ensemble infini. Soit $f : \mathbb{N} \to A$ une surjection. Montrez qu'il existe une bijection $g : \mathbb{N} \to A$.

Exercice 1.2.3

Soit A un ensemble. Soit $f : \mathbb{N} \to A$ une bijection. Soit $B \subseteq A$ un sous-ensemble infini de A. Montrez qu'il existe une bijection $g : \mathbb{N} \to B$.

Théorème 1.2.4 (Cantor):
L'ensemble des nombres réels est indénombrable.

PREUVE: On raisonne par l'absurde. Supposons au contraire qu'il existe une bijection $f : \mathbb{N} \to \mathbb{R}$. Nous allons construire un nombre réel $R \in \mathbb{R}$ qui n'est pas dans l'image de f. On définie simplement R de la manière suivante : Pour tout n, si la n-ème décimale de $f(n)$ est différente de 0 alors la n-ème décimale de R est égale à 0. A l'inverse, si la n-ème décimale de $f(n)$ est égale à 0 alors la n-ème décimale de R est égale à 1.

Il est claire que pour tout entier n, notre nombre réel R ne peut être égale à $f(n)$, car la n-ième décimale de R est différente de la n-ème décimale de $f(n)$. ∎

Que nous dis le théorème précédent ? qu'il y a "plus" de nombres réels que de nombres entiers. Les deux ensembles de nombres sont tous les deux infinis, mais l'infini des nombres réels est "plus grand" que celui des nombres entiers. En effet, quand on essaye de faire correspondre un nombre entier à chaque nombre réel, on n'aperçoit qu'il y a toujours des réels "en trop", qui ne correspondent à aucun entier.

En quoi le théorème de Cantor peut-il bien nous être utile ? Il va nous fournir notre premier argument de l'existence de nombres réels incalculables. Nous utilisons pour la proposition suivante la notion pour le moment informelle de programme informatique.

Proposition 1.2.5 : L'ensemble des programmes informatiques est dénombrable. ⋆

PREUVE: Un programme informatique (écrit dans quelque langage de programmation que ce soit) est en particulier une suite finie de caractères, où chaque caractère appartient à un alphabet fini (disons l'ensemble des caractères de la table ascii). Montrons qu'il existe une bijection de $\mathbb{N}$ vers l'ensemble des suites finies de caractères ascii. Il suffit pour cela de définir une liste de toutes les suites finies de caractères ascii : On liste d'abord toutes les suites comportant un caractère ascii, puis toutes les suites comportant deux caractères ascii, puis toutes celles en comportant trois, etc... Il est claire que n'importe quelle suite finie de caractères ascii apparaît quelque part dans notre liste.

On définie alors $f(n)$ comme étant de n-ième élément de notre liste. Il est claire que f est une bijection. En particulier l'ensemble des programmes informatiques, qui est un sous-ensemble infini des suites finies de caractères ascii, est lui aussi dénombrable, en utilisant l'exercice 1.2.3.

Théorème 1.2.6:
Il existe des nombres réels qui ne sont pas calculables.

PREUVE: Supposons que tout réel est calculé par un programme informatique. Soit P l'ensemble des programmes informatiques qui calculent un réel. Cela définit en particulier une surjection $g : P \to \mathbb{R}$. La surjection est définie simplement par $g(p) = R$ où R est le réel calculé par P.

Par la proposition 1.2.5 il y a aussi une bijection de $\mathbb{N}$ vers l'ensemble les programmes informatiques, et donc aussi une bijection f de $\mathbb{N}$ vers P qui est un sous-ensemble infini de tous les programmes informatiques. On en déduit que la fonction $n \mapsto g(f(n))$ est une surjection de $\mathbb{N}$ vers les l'ensemble réels. L'exercice 1.2.2 nous donne une bijection de $\mathbb{N}$ vers les réels, ce qui contredit le théorème 1.2.5. ∎

La preuve du théorème 1.2.6 est non-constructive : on montre l'existence de nombres incalculables sans en donner d'exemples précis. Ce sera évidement fait à maintes reprises dans la suite du cours, à travers une étude détaillée de différents types de nombres incalculables. Mais pour le moment, nous avons besoin de donner une définition mathématique précise de la notion d'être calculable. Nous introduisons pour cela un premier modèle de calcul, qui se rapproche d'un langage assembleur simplifié.

1.3 Machines à registres

1.3.1 Définitions

On va formaliser une notion de machine théorique très simple, les machines à registres. Ces machines ont une mémoire constituée d'un nombre fini de registres $R_0, R_1, \ldots, R_k$. Chaque registre est de taille non bornée et peut donc contenir un entier arbitraire positif ou nul. Elles disposent d'autre part :

— D'un programme qui est une suite finie constituée de 5 types d'instruction. Les instructions étant numérotées de 1 en 1 à partir de 0.

— D'un index de lecture qui indique l'instruction du programme à exécuter. C'est un entier inférieur à la longueur du programme.

Les différentes instructions possibles sont les suivantes :

1. Incrémenter de 1 le registre numéro i, passer à l'instruction suivante : $R_i := R_i + 1$
2. Décrémenter de 1 le registre numéro i (sauf si celui-ci est égale à 0), passer à l'instruction suivante : $R_i := R_i - 1$
3. Exécuter un goto, c'est à dire aller à l'instruction numéro p : goto p
4. Exécuter un goto conditionnel, si le registre numéro i est nul, aller à l'instruction numéro p. Sinon aller à l'instruction suivante : if $R_i = 0$ goto p
5. Une instruction d'arrêt qui apparaît une et une seule fois en fin du programme : halt.

On considère que la numérotation des instructions est implicite : le numéro désigne la place de l'instruction dans le programme, même si dans les exemples, on l'explicitera pour plus de clarté. Une telle suite d'instructions sera appelée "programme goto" dans la suite du cours. Le calcul d'un programme goto peut ne pas terminer, et les instructions `goto` et `goto` conditionnelles sont les seules instructions susceptibles de conduire le calcul à ne pas terminer.

On peut dès à présent donner une définition formelle de ce que l'on entend par fonction ou ensemble calculable.

Définition 1.3.1 : Une fonction (éventuellement partielle) $f : \mathbb{N}^n \to \mathbb{N}$ est calculable si il existe un programme goto P_f, tel que pour tout couples $x_1, \ldots, x_n$, si P_f est exécuté avec les registres $R_1, \ldots, R_n$ initialisés respectivement à $x_1, \ldots, x_n$, alors le programme s'arrête si et seulement si $f(x_1, \ldots, x_n) \downarrow y$, et alors le programme termine avec $R_0 = y$.◊

Définition 1.3.2 : Pour $n \in \mathbb{N}^*$, un sous-ensemble $X \subseteq \mathbb{N}^n$ est calculable si jamais la fonction f définie par :

$$\begin{array}{lcll} f(x_1, \ldots, x_n) & = & 1 & \text{si } (x_1, \ldots, x_n) \in X \\ f(x_1, \ldots, x_n) & = & 0 & \text{sinon} \end{array}$$

est calculable. ◊

Notez qu'un programme goto n'a qu'un nombre fini d'instructions et ne peut donc utiliser qu'un nombre fini de registres : Pour tout programme goto, il existe k tel que le programme utilise au plus les registres $R_0, \ldots, R_k$. Le programme devra donc être exécuté sur une machine à registre ayant au moins k registres.

Voyons un petit exemple de programme simple réalisé dans le cadre des machines à registres :

Exemple 1.3.3 : Le programme suivant utilise les registres R_0, R_1, R_2, R_3 pour calculer la fonction d'addition. A la fin du programme, R_0 contient le résultat de $R_1 + R_2$:

```
0  if R1 = 0 goto 4
1  R1 := R1 − 1
2  R0 := R0 + 1
3  goto 0
4  if R2 = 0 goto 8
5  R2 := R2 − 1
6  R0 := R0 + 1
7  goto 4
8  halt
```

◊

Exercice 1.3.4

Écrivez des programmes goto qui :

1. ne s'arrête pas quelque soit la valeur des registres au début de l'exécution
2. s'arrête ssi dans le registre R_1 se trouve un nombre inférieur ou égale à 2
3. calcule la fonction $a \mapsto 2a$
4. calcule la fonction
$$\begin{array}{rcll} a & \mapsto & 0 & \text{si } a = 0 \\ & \mapsto & 1 & \text{si } a > 0 \end{array}$$
5. calcule la fonction
$$\begin{array}{rcll} (a,b) & \mapsto & 1 & \text{si } a \leq b \\ & \mapsto & 0 & \text{si } a > b \end{array}$$

1.3.2 Pertinence du modèle

Notez que nous nous sommes également affranchi des contraintes pratiques des machines de la vie de tous les jours : la taille de chaque registre, le nombre de registres, et la taille du programme ne sont pas bornées. Nous renvoyons le lecteur à la section 1.2.1 où nous argumentons sur la nécessité de procéder ainsi pour notre étude théorique.

Le jeu d'instruction des machines à registres ressemble au jeu d'instruction d'un langage assembleur, mais en simplifié. Malgré cette simplicité, ce jeu d'instruction permet d'avoir autant de puissance de calcul que celle fournit par n'importe quel langage assembleur. C'est ce que nous nous efforcerons de montrer dans le reste de cette section. Nous identifions quatre types d'instructions qui reviennent dans la plupart des langages de programmation (C, java, python, etc...)

1. Les instructions d'affectation : On sauvegarde une valeur dans une variable.
2. Les instructions conditionnelles : Les instructions de type if... then... else...
3. Les instructions de boucles for : Nous entendons ici les boucles qui s'exécutent un nombre de fois déterminé au début de la boucle. Typiquement les instructions C ou

java du type `for (i = 0; i < N; i++)` pour une variable `N` (où i et N ne sont pas modifiés à l'intérieur de la boucle).

4. Les instructions de boucles `while` : Nous entendons ici les boucles plus générales, pour lesquelles on ne peut pas nécessairement prévoir le nombre de fois qu'elles s'exécutent. Typiquement les instructions C ou Java du type :
`while (!list.isEmpty())`.

Digression

Le lecteur attentif a certainement remarqué que les boucles de type `for` peuvent être simulées par des boucles de type `while`. Dès lors pourquoi les distinguer et ne pas parler uniquement des boucles de type `while` ? La raison est que cette distinction a une grande importance théorique. En particulier, si toute boucle de type `while` peut être remplacée par une boucle de type `for`, l'inverse n'est pas vrai.

Ainsi, il existe un programme P qui contient une boucle `while` qui se termine quelque soit l'entrée de P, mais tel que la boucle `while` de P fasse des choses suffisamment compliquées pour qu'il ne soit pas possible de calculer à l'avance (par un programme n'utilisant pas de boucles `while`), le nombre d'itérations qu'elle effectuera. La compréhension de cette distinction fondamentale constituera une étape importante de notre étude théorique.

Pour le moment bien sûr, attachons-nous d'abord à montrer que notre modèle de calcul des machines à registres peut faire ce qu'on attend de lui.

Afin de simuler les instructions (1) (2) (3) et (4) présentent dans les programmes informatiques usuels, on modifie le jeu d'instruction des machines à registre. On défini les programmes structurés comme étant une suite finie d'instructions parmi les instructions suivantes :

1. Une instruction parmi les suivantes déjà connues : $R_i := R_i + 1$, $R_i := R_i - 1$ et halt.
2. L'affectation de l'entier n au registre numéro i : $R_i := n$
3. L'affectation du registre j au registre numéro i : $R_i := R_j$
4. Pour toute suite finie d'instructions structurées $S = (S_0, \ldots, S_n)$ et $S' = (S'_0, \ldots, S'_m)$, l'instruction conditionnelle : if $R_i = 0$ then S else S'.
Chaque instruction de S est exécutée séquentiellement si R_i est égal à 0. Sinon chaque instruction de S' est exécutée séquentiellement.
5. Pour toute suite finie d'instructions structurées $S = (S_0, \ldots, S_n)$, l'instruction de boucle for : for $i = 0$ to R_i do S.
Soit N le nombre présent dans le registre R_i au moment où le programme commence cette instruction. Chaque instruction de S est exécutée séquentiellement, le tout N fois. Notez que si la valeur de R_i change pendant l'exécution des instructions de S, cela ne change pas le nombre de fois que la boucle s'effectue.
6. Pour toute suite finie d'instructions structurées $S = (S_0, \ldots, S_n)$, l'instruction de boucle while : while $R_i \neq 0$ do S.
Chaque instruction de S est exécutée séquentiellement tant que le registre S_i est différent de 0.

Notez qu'un programme structuré ne contient plus d'instruction `goto` ou `goto` conditionnel.

Notation

L'instruction halt est moins utile pour les programmes structurés, en particulier aucun "goto" ne peut y faire référence. On omettra donc en général cette instruction que l'on considérera présente implicitement en fin de programme.

Exemple 1.3.5 : Soit les suites d'instructions $S = (S_0, \ldots, S_n)$ et $S' = (S'_0, \ldots, S'_m)$. Une instruction de la forme "if $R_i = 0$ then S else S'" s'écriera comme suit :

Exemple

if $R_i = 0$ **then**
  S_0
  $\ldots$
  S_n
else
  S'_0
  $\ldots$
  S'_n
end

On introduisons à présent un concept qui sera utile pour faire des démonstrations par induction sur les programmes structurés.

Définition 1.3.6 : Le nombre d'imbrications maximum d'instructions de type "if then else", "for" ou "while", au sein d'un programme structuré P est appelé la *hauteur* de P.◇

Ainsi un programme P a une hauteur de 0 si il ne contient aucune instruction de type "if then else", "for" ou "while". Il a une hauteur de $n+1$ si :

1. Il contient au moins une instruction de type "if then else", "for" ou "while" pour laquelle la ou les suites d'instructions $S = (S_0, \ldots, S_n)$ correspondantes constituent un programme structuré de hauteur n.
2. Pour toutes les instructions de type "if then else", "for" ou "while", la ou les suites d'instructions $S = (S_0, \ldots, S_n)$ correspondantes constituent un programme structuré de hauteur au plus n.

Montrons à présent que si un programme structuré calcule quelque chose, alors il existe un programme goto qui calcule la même chose. Nous montrerons plus tard avec le corollaire 2.4.6 que l'inverse est aussi vrai : si un programme goto calcule quelque chose, alors il existe un programme structuré qui calcule la même chose.

Proposition 1.3.7 : Si une fonction $f : \mathbb{N}^n \to \mathbb{N}$ est calculable par un programme structuré P, alors elle est calculable par un programme goto M. ⋆

Preuve: Afin de montrer la proposition, il nous faut considérer une variante minime des programmes structurés : des programmes pour lesquels une d'instructions $S = (S_0, \ldots, S_n)$ peut contenir des `goto` et `goto` conditionnels qui font référence à une des instructions de la séquence (une instruction "S_j =`goto i`" signifiera que l'on passe de l'instruction S_j de la séquence à l'instruction S_i de la séquence). On considère pour cela les séquences d'instructions faisant parti d'une instruction "if then else", "for" ou "while". On considère aussi que les instructions du programme lui-même, forment une séquence d'instructions (et donc des instructions `goto` du programme peuvent faire référence à d'autres instructions du programme).

Appelons ces programmes des programmes structurés étendus. Montrons à présent que pour un programme structuré (éventuellement étendu) de hauteur $N+1$, il est possible d'obtenir un programme structuré étendu de hauteur N qui calcule la même chose.

Considérons pour cela deux suites d'instructions structurées étendus $S = (S_0, \ldots, S_n)$ et $S' = (S'_0, \ldots, S'_n)$ tel que S et S' ne contiennent aucune instruction de type "if then else", "for" ou "while" (de manière équivalente, les suites d'instructions S et S' sont des programmes structurés étendu de hauteur 0).

Montrons que pour les trois types d'instructions "if $R_i = 0$ then S else S'", "for $i = 1$ to R_i do S" et "while $R_i \neq 0$ do S", on peut obtenir une suite d'instructions étendus équivalente, sans "if then else", "for" ou "while".

1. Considérons l'instruction conditionnelle suivante :

   a if $R_i = 0$ then S else S'

   Le numéro "a" a été rajouté, et correspond à la place de l'instruction dans la suite d'instructions dont elle fait parti. On peut alors la remplacer par les instructions suivantes :

   $$\begin{aligned}
   a &\quad \text{if } R_i = 0 \text{ goto } a+1+1 \\
   a+1 &\quad \text{goto } a+1+n+1+1 \\
   a+1+1 &\quad S_1 \\
   &\quad \ldots \\
   a+1+n &\quad S_n \\
   a+1+n+1 &\quad \text{goto } a+1+n+1+m+1 \\
   a+1+n+1+1 &\quad S'_1 \\
   &\quad \ldots \\
   a+1+n+1+m &\quad S'_m
   \end{aligned}$$

   Puis dans les instructions suivantes, toutes les instructions "goto k" pour $k > a$ sont remplacées par des instructions "goto $a+1+n+1+m$".

2. Supposons que dans le programme utilise au plus les registres $R_1, \ldots, R_k$. Considérons l'instruction de boucle `for` suivante :

   a for $i = 1$ to R_i do S

   Le numéro "a" a été rajouté, et correspond à la place de l'instruction dans la suite d'instructions dont elle fait parti. On peut alors la remplacer par les instructions suivantes :

   $$\begin{aligned}
   a &\quad R_{k+1} = R_i \\
   a+1 &\quad R_{k+1} = R_{k+1} - 1 \\
   a+2 &\quad \text{if } R_{k+1} = 0 \text{ goto } a+2+n+2 \\
   a+2+1 &\quad S_1 \\
   &\quad \ldots \\
   a+2+n &\quad S_n \\
   a+2+n+1 &\quad \text{goto } a+1
   \end{aligned}$$

   Puis dans la suite du programme, toutes les instructions "goto k" pour $k > a$ sont remplacées par des instructions "goto $a+1+n+1+m$".

3. Considérons enfin l'instruction de boucle `while` suivante :

   a while $R_i \neq 0$ do S

   Le numéro "a" a été rajouté, et correspond à la place de l'instruction dans la suite d'instructions dont elle fait parti. On peut alors la remplacer par les instructions suivantes :

$$\begin{array}{rl} a & \text{if } R_i = 0 \text{ goto } a+n+2 \\ a+1 & S_1 \\ & \dots \\ a+n & S_n \\ a+n+1 & \text{goto } a \end{array}$$

En utilisant (1) (2) et (3), on voit comment passer d'un programme structuré étendu de hauteur $N+1$ à un programme structuré étendu de hauteur N. En répétant l'opération autant de fois que nécessaire, on peut toujours obtenir un programme structuré étendu de hauteur 0, c'est à dire sans instruction de type "if then else", "for" ou "while".

Tout ce qu'il reste à présent à faire, c'est de montrer que l'on peut passer d'un programme structuré étendu de hauteur 0, à un programme goto. Montrons d'abord que l'on peut toujours se passer de l'instruction d'affectation d'un entier à un registre. Supposons que dans le programme P, on ait l'instruction suivante :

$a \quad R_i := n$

On la remplace alors par les instructions :

$$\begin{array}{rl} a & \text{if } R_i = 0 \text{ goto } a+3 \\ a+1 & R_i := R_i - 1 \\ a+2 & \text{goto } a \\ a+2+1 & R_i := R_i + 1 \\ & \dots \\ a+2+N & R_i := R_i + 1 \end{array}$$

Puis dans la suite du programme, toutes les instructions "goto k" pour $k > a$ sont remplacées par des instructions "goto $k+N+2$".

Montrons maintenant que l'on peut toujours se passer de l'instruction d'affectation d'un registre à un registre. Supposons que dans le programme P utilisant au plus les registres $R_1, \dots, R_k$, on ait l'instruction suivante :

$a \quad R_i := R_j$

On la remplace alors par les instructions :

$$\begin{array}{rl} a & R_i := 0 \\ a+1 & \text{if } R_j = 0 \text{ goto } a+6 \\ a+2 & R_i := R_i + 1 \\ a+3 & R_j := R_j - 1 \\ a+4 & R_{k+1} := R_{k+1} + 1 \\ a+5 & \text{goto } a+1 \\ a+6 & \text{if } R_{k+1} = 0 \text{ goto } a+10 \\ a+7 & R_{k+1} := R_{k+1} - 1 \\ a+8 & R_j := R_j + 1 \\ a+9 & \text{goto } a+6 \end{array}$$

Puis dans la suite du programme, toutes les instructions "goto k" pour $k > a$ sont remplacées par des instructions "goto $k+10$". ■

Exercice 1.3.8

Écrivez un programme structuré qui :

1. calcule la fonction qui a (a, b) associe $a \times b$

2. calcule la fonction qui a a associe le plus grand entier b tel que $b^2 \leq a$
3. calcule la fonction qui a a associe le plus petit entier b tel que $a \leq 2^b$

Exercice 1.3.9

Ecrivez un programme structuré qui s'arrête ssi le nombre dans le registre R_1 est pair.

Chapitre 2

Les fonctions récursives

Nous allons à présent étudier un autre modèle de calcul, qui nous permettra de conduire une étude plus mathématique de l'étude des objets calculable et incalculables : les fonctions récursives et primitives récursives.

2.1 Les fonctions primitives récursives

Nous allons définir les fonctions primitives récursives. Nous aurons besoin pour cela de la notion de projection :

Définition 2.1.1 : Pour $n \in \mathbb{N}$ et a avec $1 \leq a \leq n$, la fonction constante $c_a^n : \mathbb{N}^n \to \mathbb{N}$ est définie par $c_a^n(x_1, \ldots, x_n) = a$. ◊

Définition 2.1.2 : Pour $n \in \mathbb{N}^*$ et i avec $1 \leq i \leq n$, la fonction de projection $p_i^n : \mathbb{N}^n \to \mathbb{N}$ est définie par $p_i^n(x_1, \ldots, x_i, \ldots, x_n) = x_i$. ◊

Définition 2.1.3 : L'ensemble des fonctions primitives récursives est le plus petit ensemble $E \subseteq \bigcup_{n \in \mathbb{N}^*} \mathbb{N}^{(\mathbb{N}^n)}$ tel que :

- E contient toutes les fonctions constantes $c_a^n : \mathbb{N}^n \to \mathbb{N}$ pour tout $n \in \mathbb{N}$.
- E contient toutes les projections $p_i^n : \mathbb{N}^n \to \mathbb{N}$ pour tout $n \in \mathbb{N}^*$ et i avec $1 \leq i \leq n$.
- E contient la fonction successeur $succ : \mathbb{N} \to \mathbb{N}$, définie par $succ(n) = n + 1$.
- E est clôt par le schéma de composition : Pour tout $n, m \in \mathbb{N}^*$, si :
  - La fonction $g : \mathbb{N}^n \to \mathbb{N}$ appartient à E
  - les fonctions $h_1, \ldots, h_n : \mathbb{N}^m \to \mathbb{N}$ appartiennent à E

  alors la fonction $f : \mathbb{N}^m \to \mathbb{N}$ définie par

  $$f(x_1, \ldots, x_m) = g(h_1(x_1, \ldots, x_m), \ldots, h_n(x_1, \ldots, x_m))$$

  appartient à E.
- E est clôt par le schéma de récurrence : pour tout n, si :
  - La fonction $g : \mathbb{N}^n \to \mathbb{N}$ appartient à E
  - La fonction $h : \mathbb{N}^{n+2} \to$ appartient à E

alors la fonction $f : \mathbb{N}^{n+1} \to \mathbb{N}$ définie par :

$$\begin{aligned} f(x_1, \ldots, x_n, 0) &= g(x_1, \ldots, x_n) \\ f(x_1, \ldots, x_n, y+1) &= h(x_1, \ldots, x_n, y, f(x_1, \ldots, x_n, y)) \end{aligned}$$

appartient à E. ◊

De la même manière que l'on a défini les sous-ensemble calculable de $\mathbb{N}^n$, on défini aussi les sous-ensemble de $\mathbb{N}^n$ qui sont primitifs récursifs :

Définition 2.1.4 : Pour $n \in \mathbb{N}^*$, un sous-ensemble $X \subseteq \mathbb{N}^n$ est primitif récursif si la fonction f définie par :

$$\begin{aligned} f(x_1, \ldots, x_n) &= 1 \quad \text{si } (x_1, \ldots, x_n) \in X \\ f(x_1, \ldots, x_n) &= 0 \quad \text{sinon} \end{aligned}$$

est primitive récursive. ◊

Notation

On appellera parfois les ensembles $X \subseteq \mathbb{N}^n$ des *prédicats*. Ils sont alors vu comme une valeur de vérité. Ainsi si $P \subseteq \mathbb{N}^n$ est un prédicat, on écriera $P(x_1, \ldots, x_n)$ pour signifier $(x_1, \ldots, x_n) \in P$.

Les fonctions primitives récursives contiennent déjà un très grand nombre de fonctions. Les but des exercices suivants est de vous en convaincre, et d'apprendre à les manipuler :

Exercice 2.1.5

Montrez que les fonctions suivantes sont primitives récursives (réutilisez les fonctions de chaque question pour la question suivante) :

1. La fonction $s_2 : x \mapsto x + 2$
2. La fonction $sp_2 : (x, y) \mapsto y + 2$
3. La fonction $t_2 : x \mapsto 2x$
4. La fonction $f : x \mapsto 2x + 1$

Exercice 2.1.6

Montrez que l'ensemble des fonctions primitives récursives est clôt par le schéma de définition par itération : pour une fonction $g : \mathbb{N}^p \to \mathbb{N}$, et pour une fonction $h : \mathbb{N}^{p+1} \to \mathbb{N}$, la fonction

$$\begin{aligned} f(x_1, \ldots, x_p, 0) &= g(x_1, \ldots, x_p, 0) \\ f(x_1, \ldots, x_p, n+1) &= h(x_1, \ldots, x_p, f(x_1, \ldots, x_p, n)) \end{aligned}$$

est primitive récursive.

Exercice 2.1.7

Montrez que l'addition, la multiplication et la fonction exponentielle sont primitives récursives

Exercice 2.1.8

Montrez que les fonctions primitives récursives sont closes par schéma de composition étendu. Soit $n, m \in \mathbb{N}^*$. Soit $g : \mathbb{N}^n \to \mathbb{N}$ primitive récursive. Pour tout $i \leq n$ soit $m_i \leq m$, soit $h_i : \mathbb{N}^{m_i} \to \mathbb{N}$ primitive récursive et soit $1 \leq a_1^i < \cdots < a_{m_i}^i \leq m$ des indices. Alors la fonction $f : \mathbb{N}^m \to \mathbb{N}$ définie par :

$$f(x_1, \ldots, x_m) = g(h_1(x_{a_1^1}, \ldots, x_{a_{m_1}^1}), \ldots, h_n(x_{a_1^n}, \ldots, x_{a_{m_n}^n}))$$

est primitive récursive.

Exercice 2.1.9

Montrez que la fonction $sg : \mathbb{N} \to \mathbb{N}$ qui à 0 associe 0 et qui à tous les autres entiers associe 1, est primitive récursive. Montrez que la fonction $\overline{sg} : \mathbb{N} \to \mathbb{N}$ qui à 0 associe 1 et qui à tous les autres entiers associe 0, est primitive récursive.

Exercice 2.1.10

Montrez que les prédicats de comparaisons $\leq, <, \geq, >, =, \neq$ sont primitifs récursifs.

Exercice 2.1.11

Montrez que si g est une fonction primitive récursive de $\mathbb{N}^{p+1}$ dans $\mathbb{N}$, alors les fonctions :

$$f_1(x_1, \ldots, x_p, n) = \sum_{i=0}^{n} g(x_1, \ldots, x_p, i)$$

et

$$f_2(x_1, \ldots, x_p, n) = \prod_{i=0}^{n} g(x_1, \ldots, x_p, i)$$

sont primitives récursives.

Exercice 2.1.12

Montrez que les fonctions :

1. $pred : \mathbb{N} \to \mathbb{N}$ qui vaut 0 en 0 et $n-1$ en $n > 0$.
2. $- : \mathbb{N}^2 \to \mathbb{N}$ qui vaut $\max(0, a-b)$

sont primitives récursives.

Exercice 2.1.13

Soit $P_1(a_1, \ldots, a_n)$ et $P_2(b_1, \ldots, b_m)$ des prédicats primitifs récursifs.

1. Montrez que le prédicat $P(a_1, \ldots, a_n, b_1, \ldots, b_m)$ qui est vrai ssi $P_1(a_1, \ldots, a_n)$ est vrai et $P_2(b_1, \ldots, b_m)$ est vrai , est primitif récursif.
2. Montrez que le prédicat $P(a_1, \ldots, a_n, b_1, \ldots, b_m)$ qui est vrai ssi $P_1(a_1, \ldots, a_n)$

est vrai ou $P_2(b_1, \ldots, b_m)$ est vrai , est primitif récursif.

3. Montrez que le prédicat $P(a_1, \ldots, a_n)$ qui est vrai ssi $P_1(a_1, \ldots, a_n)$ est faux, est primitif récursif.

Exercice 2.1.14

Montrez que les prédicats primitifs récursifs sont clôt par quantification existentielle et universelle bornée : Si $P(x_1, \ldots, x_n, z)$ est un prédicat primitif récursif, alors les prédicats :

$$\begin{aligned} Q_1(x_1, \ldots, x_n, z) &= \exists t \leq z \; P(x_1, \ldots, x_n, z) \\ Q_2(x_1, \ldots, x_n, z) &= \forall t \leq z \; P(x_1, \ldots, x_n, z) \end{aligned}$$

sont primitifs récursifs.

Exercice 2.1.15

Montrez que l'ensemble des fonctions primitives récursives est clôt par définition par cas sur un prédicat primitif récursif : si g et h sont deux fonction primitives récursives de $\mathbb{N}^p$ dans $\mathbb{N}$, et si P est un prédicat primitif récursif sur $\mathbb{N}^p$, alors la fonction :

$$\begin{aligned} f(x_1, \ldots, x_p) &= g(x_1, \ldots, x_p) && \text{si } P(x_1, \ldots, x_p) \\ f(x_1, \ldots, x_p) &= h(x_1, \ldots, x_p) && \text{sinon} \end{aligned}$$

est primitive récursive.

Exercice 2.1.16

Montrez que les fonctions primitives récursives sont closes par minimisation bornée : si $f : \mathbb{N}^{p+1} \to \mathbb{N}$ est primitive récursive, alors la fonction

$$\begin{aligned} g(x_1, \ldots, x_p, n) &= \min\{t \leq n \mid f(x_1, \ldots, x_p, t) = 0\} + 1 && \text{si } \exists t \leq n \; f(x_1, \ldots, x_p, t) = 0 \\ &= 0 && \text{sinon} \end{aligned}$$

est primitive récursive.

2.2 Fonctions récursives

Les fonctions primitives récursives ne suffisent pas à capturer toutes les fonctions calculables. Nous verrons en fait que les fonctions primitives récursives sont exactement les fonctions calculable par des programmes structurés qui n'utilisent pas de boucles `while`. Afin de capturer l'ensemble de toutes les fonctions calculables, il nous faut définir d'autres fonctions, et on définit pour cela le schéma suivant :

Définition 2.2.1 : Étant donnée une fonction $f : \mathbb{N}^{n+1} \to \mathbb{N}$, on écrit $\mu z.f(z, x_1, \ldots, x_n) = 0$ la fonction partielle de $\mathbb{N}^n \to \mathbb{N}$ qui à $x_1, \ldots, x_n$ associe le plus petit z tel que $f(z, x_1, \ldots, x_n) = 0$ si une telle valeur z existe. Sinon la fonction $\mu z.f(z, x_1, \ldots, x_n) = 0$ n'est pas définie et on écriera $\mu z.f(z, x_1, \ldots, x_n) = 0 \uparrow$. ◊

Définition 2.2.2 : L'ensemble des fonctions récursives partielles est le plus petit ensemble $E \subseteq \bigcup_{n\in\mathbb{N}^*} \mathbb{N}^{(\mathbb{N}^n)}$ tel que :

- E contient toutes les fonctions primitives récursives.
- E est clôt par le schéma de composition.
- E est clôt par le schéma de récurrence.
- E est clôt par le schéma de minimisation μ : si $f : \mathbb{N}^{n+1} \to \mathbb{N}$ appartient à E, alors la fonction $\mu z.f(z, x_1, \ldots, x_n) = 0$ appartient à E.

Une fonction récursive partielle qui est définie partout est appelée simplement fonction récursive, ou encore fonction récursive totale. ♢

Notez que l'on autorise l'application des schémas de composition, récurrence et minimisation sur des fonctions qui ne sont éventuellement pas définies partout. Dans ces cas là la partialité "se propage" comme attendu. Nous donnons l'exemple pour le schéma de composition. Pour $h : \mathbb{N}^n \to \mathbb{N}$ et pour $g_1 : \mathbb{N}^m \to \mathbb{N}$, ..., $g_n : \mathbb{N}^m \to \mathbb{N}$ des fonctions récursives partielles, la fonction $f : \mathbb{N}^m \to \mathbb{N}$ telle que :

$$\begin{array}{rcll} f(x_1,\ldots,x_m) & = & h(g_1(x_1,\ldots,x_m),\ldots,g_n(x_1,\ldots,x_m)) & \text{si } \forall i \leq n\ g_i(x_1,\ldots,x_m)\downarrow \\ & = & \uparrow & \text{sinon} \end{array}$$

est une fonction récursive partielle. Il en va de même pour le schéma de récurrence et celui de minimisation.

Il existe une autre manière équivalente de définir les fonctions récursives. On définie E_0 comme l'ensemble de toutes les fonctions constantes, les projections et la fonction successeur. On définie ensuite inductivement E_{n+1} comme étant la clôture de E_n par le schéma de composition, par le schéma de récurrence, et par le schéma de minimisation μ. L'ensemble E des fonctions récursives est alors égale à $\bigcup_n E_n$. Pour une fonction $f \in E$, on dira que n est *la hauteur* de E si n est le plus petit tel que $f \in E_n$. Notez que la hauteur est définie aussi bien pour les fonctions primitives récursives que pour les fonctions récursives. Simplement pour définir la hauteur d'une fonction primitive récursive, on ne tient pas compte de la clôture par le schéma μ.

Exercice 2.2.3

Montrez que pour tout prédicat $P \subseteq \mathbb{N}^{k+1}$, la fonction suivante est récursive :

$$\begin{array}{rcll} f(x_1,\ldots,x_k) & = & \min\{z\ :\ P(z,x_1,\ldots,x_k)\} & \text{si } \exists z\ P(z,x_1,\ldots,x_k) \\ & = & \text{non défini} & \text{sinon} \end{array}$$

Exercice 2.2.4

Montrez que pour tout prédicat $P \subseteq \mathbb{N}^{k+1}$ et pour toute fonction récursive $g : \mathbb{N}^k \to \mathbb{N}$, la fonction suivante est récursive :

$$\begin{array}{rcll} f(x_1,\ldots,x_k) & = & g(x_1,\ldots,x_k) & \text{si } \exists z\ P(z,x_1,\ldots,x_k) \\ & = & \text{non défini} & \text{sinon} \end{array}$$

Exercice 2.2.5

Soit $a_1, \ldots, a_k \in \mathbb{N}$. Montrez que la fonction nulle en $a_1, \ldots, a_k$ et non définie ailleurs, est récursive.

Exercice 2.2.6

Montrez que la fonction pre, non définit en 0 et vérifiant $\text{pre}(x+1) = x$ est récursive.

2.3 Les fonctions récursives sont calculables

Nous allons montrer ici que toutes les fonctions récursives sont aussi calculables par des programmes structurés, et donc aussi par des programmes goto. Afin de faciliter les preuves à venir, nous introduisons ici la notion de programme propre :

Définition 2.3.1 : Un programme propre est un programme structuré qui termine son calcul avec tous ses registres, à l'exception de son registre de sortie R_0, dans le même état qu'au début du calcul. ♢

Proposition 2.3.2 : Pour tout programme structuré, il existe un programme structuré propre qui a le même comportement. ⋆

PREUVE: Soit m tel que notre programme utilise au plus les registres $R_0, \ldots, R_m$. Il suffit de recopier au début du programme les registres de R_1 à R_m dans les registres de R_{m+1} à R_{m+m}. Ensuite le programme est le même en remplaçant tous les registres R_i par R_{i+m} pour $i \geq 0$, puis enfin en rajoutant des instructions à la fin qui remettent à zéro tous les registres de R_{m+1} à R_{m+m}. ■

La première partie de la preuve que les fonctions récursives de base sont calculables par des programmes structurés est simple, et on la donne en exercice.

Exercice 2.3.3

Montrez que les fonctions constantes, tout comme les fonctions de projections et la fonction successeur sont calculables par des programmes structurés (et donc par des programmes `goto`).

Théorème 2.3.4:
Toute fonction récursive partielle est calculable par un programme structuré (et donc par un programme `goto`).
Toute fonction primitive récursive est calculable par un programme structuré sans boucles "while".

PREUVE: La preuve est faite par induction sur la hauteur d'une fonction récursive ou primitive récursive. Si f a une hauteur de 0, alors se référer à l'exercice 2.3.3. Supposons maintenant par induction que le théorème est vérifié pour les fonctions récursives et primitives récursives de hauteur n. Soit f une fonction récursive de hauteur $n+1$. On a les cas suivants :

1. Schéma de composition : supposons que

$$f(x_1, \ldots, x_m) = g(h_1(x_1, \ldots, x_m), \ldots, h_k(x_1, \ldots, x_m))$$

pour des fonctions $g, h_1, \ldots, h_k$ de hauteurs inférieures ou égales à n. En particulier il y a des programmes structurés propres $F, H_1, \ldots, H_m$ qui calculent chacune de ces fonctions. Supposons que chacun de ces programmes utilisent au plus les registres $R_0, \ldots, R_z$ pour $z > m$. Le programme pour calculer F est donné simplement par la suite d'instructions suivante :

Programme F
⟨Instructions de H_1⟩
$R_{z+1} := R_0$
⟨Instructions de H_2⟩
$R_{z+2} := R_0$
...
⟨Instructions de H_k⟩
$R_{z+k} := R_0$
$R_1 := R_{z+1}$
...
$R_k := R_{z+k}$
⟨Instructions de G⟩

Notez que si $G, H_1, \ldots H_m$ n'utilisent pas de boucles "while", alors le programme pour calculer F n'utilise pas de boucles "while" non plus.

2. Schéma de récurrence : Supposons que f est définie par

$$\begin{aligned} f(x_1, \ldots, x_n, 0) &= g(x_1, \ldots, x_n) \\ f(x_1, \ldots, x_n, y+1) &= h(x_1, \ldots, x_n, y, f(x_1, \ldots, x_n, y)) \end{aligned}$$

pour g et h de hauteurs inférieures ou égales à n. En particulier il y a des programmes structurés propres G et H pour calculer g et h, qui utilisent au plus les registres R_0 à R_z. Le programme suivant permet de calculer F :

Programme F
$R_{z+1} := R_{n+1}$
⟨Instructions de G⟩
$R_{n+1} := 1$
$R_{n+2} := R_0$
for $i = 1$ **to** R_{z+1} **do**
⟨Instructions de H⟩
$R_{n+1} := R_{n+1} + 1$
$R_{n+2} := R_0$
end

Notez que si G, H n'utilisent pas de boucles "while", alors le programme pour calculer F n'utilise pas de boucles "while" non plus.

3. Schéma μ : Supposons que f soit définie par $f(x_1, \ldots, x_n) = \mu x.g(x_1, \ldots, x_n, x) = 0$ pour g de hauteur inférieure à n. En particulier il y a un programme propre G pour calculer g, qui utilise au plus les registres R_0 à R_z. Le programme suivant permet

de calculer F :

Programme F

$R_{z+1} := 1$
$R_{n+1} := 0$
while $R_{z+1} \neq 0$ **do**
  ⟨Instructions de G⟩
  $R_{z+1} := R_0$
  $R_{n+1} := R_{n+1} + 1$
end

les points (1) (2) et (3) concluent la récurrence et donc la preuve. ∎

2.4 Les fonctions calculables sont récursives

Afin de montrer que les fonctions calculables sont récursives, on aura besoin des exercices suivants, qui permettent en particulier de coder des k-uplets et des listes d'entiers de manière récursive.

L'exercice suivant montre comment coder des couples d'entiers de manière primitive récursive.

Exercice 2.4.1

Soit $\alpha_2 : \mathbb{N}^2 \to \mathbb{N}$ la bijection de Cantor définie par :

$$\begin{aligned} \alpha_2(x,y) &= y + \textstyle\sum_{i=0}^{x+y} i \\ &= y + \frac{(x+y+1)(x+y)}{2} \end{aligned}$$

1. Montrez que α_2 est bijective.
2. Montrez que α_2 est primitive récursive.
3. Montrez que $\alpha_2(a,b) \geq a$ et $\alpha_2(a,b) \geq b$.
4. Montrez que les deux projections π_1, π_2 tel que :
   — $\alpha_2(\pi_1(c), \pi_2(c)) = c$
   — $\pi_1(\alpha_2(a,b)) = a$
   — $\pi_2(\alpha_2(a,b)) = b$ sont primitives récursives.

L'exercice suivant utilise l'exercice précédent pour coder des k-uplets d'entiers de manière primitive récursive, pour n'importe quel $k > 1$.

Exercice 2.4.2

On définie par récurrence sur $k \geq 2$ les fonctions :

$$\alpha_{k+1}(x_1, x_2, \ldots, x_{k+1}) = \alpha_2(x_1, \alpha_k(x_2, \ldots, x_{k+1}))$$

1. Montrez que pour tout $k \geq 2$ la fonction α_k est une bijection de $\mathbb{N}^k$ dans $\mathbb{N}$.
2. Montrez que pour tout $k \geq 2$ la fonction α_k est primitive récursive.

3. Montrez que pour tout $k \geq 2$ les projections $\pi_1^k, \ldots, \pi_k^k$ tels que

$$\alpha_k(\pi_1^k(c), \ldots, \pi_k^k(c)) = c$$

sont primitives récursives.

Notation

Dans la suite, on utilisera aussi $\langle n, m \rangle$ pour parler de l'entier $\alpha_2(n, m)$ qui code le couple (n, m). On utilisera de la même manière la notation $\langle x_1, \ldots, x_k \rangle$ pour parler de l'entier $\alpha_k(x_1, \ldots, x_k)$ qui code le couple $(x_1, \ldots, x_k)$.

Notation

Pour simplifier la lecture, on utilisera également pour tout k les notations $\pi_1(z), \ldots, \pi_k(z)$ pour désigner les projections $\pi_1^k(z), \ldots, \pi_k^k(z)$. Il s'agit d'un abus de notation car la fonction $\pi_1^2(z)$ est une fonction différente de $\pi_1^3(z)$. On se permet cet abus de notation car il sera toujours clair dans le contexte à laquelle de ces fonction on fait référence.

L'exercice suivant fournit un exemple de la manière dont le codage des k-uplet peut être utile. Grâce à ce codage, on peut définir des fonctions primitives récursives se faisant référence l'une l'autre, par récurrence mutuelle.

Exercice 2.4.3

Utilisez les fonctions α_k pour montrer que si les fonctions $g_1, \ldots, g_k$ de $\mathbb{N}^n$ dans $\mathbb{N}$ et les fonctions $h_1, \ldots, h_n$ de $\mathbb{N}^{n+k+1}$ dans $\mathbb{N}$ sont primitives récursives, alors les fonctions :

$$\begin{aligned}
f_1(x_1, \ldots, x_n, 0) &= g_1(x_1, \ldots, x_n) \\
f_1(x_1, \ldots, x_n, x+1) &= h_1(x_1, \ldots, x_n, x, f_1(x_1, \ldots, x_n, x), \ldots, f_k(x_1, \ldots, x_n, x)) \\
&\ldots \\
f_k(x_1, \ldots, x_n, 0) &= g_k(x_1, \ldots, x_n) \\
f_k(x_1, \ldots, x_n, x+1) &= h_k(x_1, \ldots, x_n, x, f_1(x_1, \ldots, x_n, x), \ldots, f_k(x_1, \ldots, x_n, x))
\end{aligned}$$

sont primitives récursives.

On aborde à présent l'exercice le plus important pour les démonstrations à venir. Il utilise les exercices précédents afin d'établir un codage des listes : des suites d'entiers de longueur arbitrairement grande.

Exercice 2.4.4

Soit $::$ une notation pour la fonction primitive récursive $a :: l = 1 + \alpha_2(a, l)$. Soit hd et tl les fonctions définies par :

$$\begin{aligned}
\mathrm{hd}(0) &= 0 & \qquad \mathrm{tl}(0) &= 0 \\
\mathrm{hd}(x :: l) &= x & \qquad \mathrm{tl}(x :: l) &= l
\end{aligned}$$

On définie la notation [] pour représenter le codage suivant des listes d'entiers :

$$\begin{array}{rcl} [] & = & 0 \\ [a_0, a_1, \ldots, a_n] & = & a_0 :: [a_1, \ldots, a_n] \end{array}$$

1. Montrez que la fonction [] est bijective.
2. Montrez que les fonctions hd et tl sont primitives récursives.
3. Montrez que la fonction gett : $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$ telle que $\text{gett}([x_0, \ldots, x_n], i) = [x_i, \ldots, x_n]$ est primitive récursive.
4. Montrez que la fonction get : $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$ telle que $\text{get}([x_0, \ldots, x_n], i) = x_i$ est primitive récursive.

Avant de montrer que toute fonction calculable est récursive, récapitulons ici les fonctions primitives récursives des différents exercices précédents, que nous allons utiliser :

1. La fonction $\text{pred} : \mathbb{N} \to \mathbb{N}$ telle que $\text{pred}(n) = \max(0, n-1)$
2. Le codage des n-uplet : Pour tout n, la bijection primitive récursive :

$$\langle\rangle : \mathbb{N}^n \to \mathbb{N}$$

3. Le décodage des n-uplet : Pour tout n, les n projections primitives récursives :

$$\pi_k : \mathbb{N} \to \mathbb{N} \text{ (pour } k \leq n \text{)}$$

4. Le codage des listes : Il existe une bijection $[] : \bigcup_{n \in \mathbb{N}^*} \mathbb{N}^n \to \mathbb{N}$ telle que les fonctions suivantes sont primitives récursives :
   (a) La fonction :: de $\mathbb{N}$ vers $\mathbb{N}$ telle que $a :: [x_1, \ldots, x_n] = [a, x_1, \ldots, x_n]$
   (b) La fonction $\text{hd} : \mathbb{N} \to \mathbb{N}$ telle que $\text{hd}([x_0, \ldots, x_n]) = x_1$
   (c) La fonction $\text{tl} : \mathbb{N} \to \mathbb{N}$ telle que $\text{tl}([x_0, x_1, \ldots, x_n]) = [x_1, \ldots, x_n]$
   (d) La fonction $\text{get} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ telle que $\text{get}([x_0, \ldots, x_n], i) = x_i$

On passe à présent à la preuve du théorème principale de cette section.

Théorème 2.4.5:
Tout fonction calculable par un programme `goto` *(et donc aussi par un programme structuré) est aussi une fonction récursive.*

PREUVE: Avant de faire la preuve, on va avoir besoin de fixer différents codages.

Codage des programmes goto :

On fixe le codage suivant des programmes goto. On code pour cela les instructions de la manière suivante :

— $R_i = R_i + 1$ est codé par $\langle 0, i \rangle$
— $R_i = R_i - 1$ est codé par $\langle 1, i \rangle$
— goto n est codé par $\langle 2, n \rangle$
— if $R_i = 0$ goto n est codé par $\langle 3, \langle i, n \rangle \rangle$
— halt est codé par $\langle 4, 0 \rangle$

Pour des raisons techniques, il sera utile d'avoir comme information une borne sur l'indice maximal des registres utilisés. Le code d'un programme goto est simplement donné par le code : $\langle k, [I_1, \ldots, I_n] \rangle$ où k est tel que le programme utilise au plus les registres $R_0, \ldots, R_k$, et où I_e est le code de la e-ième instruction pour $1 \leq e \leq n$.

Codage des machines à registres :

On fixe à présent un codage de l'état d'une machine à k registre. Cet état est donné par la valeur des registres ainsi que par le numéro d'instruction à exécuter. Pour un nombre de registre k donné, ce code est $\langle m, [R_0, \ldots, R_k] \rangle$ où m est le numéro d'instruction et R_i est la valeur du registre numéro i pour $1 \leq i \leq k$.

Fonction d'initialisation :

Fixons maintenant une fonction d'initialisation Init, qui étant donné un code $e = \langle k, I \rangle$ d'un programme (où I est une liste d'instructions), des valeurs $x_1, \ldots, x_n$ (pour $n \leq k$), donne le code représentant l'état de la machine à k registres, au début du calcul.

$$\text{Init}(\langle k, I \rangle, x_1, \ldots, x_n) = \langle 0, x_1 :: \ldots :: x_n :: \text{aux}(k-n) \rangle$$

avec aux définie par :

$$\begin{aligned} \text{aux}(0) &= [\,] \\ \text{aux}(k+1) &= 0 :: \text{aux}(k) \end{aligned}$$

Il est claire que la fonction Init est primitive récursive.

Fonctions de transition 1 :

On va ici créer une fonction de transition tr_1, qui étant donnée le code $e = \langle k, I \rangle$ d'un programme et le code $p = \langle m, R \rangle$ de l'état d'une machine, renvoie le numéro de la prochaine instruction à exécuter à l'étape de calcul suivante. On va utiliser pour cela une fonction $\text{cur} : \mathbb{N} \to \mathbb{N}$ qui, étant donnée le code $e = \langle k, I \rangle$ d'un programme et le code $p = \langle m, R \rangle$ de l'état d'une machine, permet d'obtenir l'instruction courante de la machine :

$$\text{cur}(\langle k, I \rangle, \langle m, R \rangle) = \text{get}(I, m)$$

On peut maintenant définir notre fonction tr_1 :

$$\begin{aligned}
&/* \text{ incrémentation } */ \\
\text{tr}_1(e,p) &= \pi_1(p) + 1 && \text{si} && \pi_1(\text{cur}(e,p)) = 0 \\
&/* \text{ décrémentation } */ \\
&= \pi_1(p) + 1 && \text{si} && \pi_1(\text{cur}(e,p)) = 1 \\
&/* \text{ goto } */ \\
&= \pi_2(\text{cur}(e,p)) && \text{si} && \pi_1(\text{cur}(e,p)) = 2 \\
&/* \text{ goto conditionel et } R_i = 0 */ \\
&= \pi_2(\pi_2(\text{cur}(e,p))) && \text{si} && \pi_1(\text{cur}(e,p)) = 3 \text{ and} \\
& && && \text{get}(\pi_2(p), \pi_1(\pi_2(\text{cur}(e,p)))) = 0 \\
&/* \text{ goto conditionel et } R_i \neq 0 */ \\
&= \pi_1(e) + 1 && \text{si} && \pi_1(\text{cur}(e,p)) = 3 \text{ and} \\
& && && \text{get}(\pi_2(p), \pi_1(\pi_2(\text{cur}(e,p)))) \neq 0 \\
&/* \textit{ halt } */ \\
&= \pi_1(e) && \text{sinon}
\end{aligned}$$

Il est claire que tr_1 est primitive récursive.

Fonctions de transition 2 :

On va à présent définir une fonction primitives récursives tr_2 qui étant donnée le code e d'un programme et le code p de l'état d'une machine, permet d'obtenir l'état des registres de la machine à l'étape de calcul suivante.

Pour cela on utilisera deux fonctions primitives récursives $\mathrm{inc} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ et $\mathrm{dec} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, telles que

$$\mathrm{inc}([x_0, \ldots, x_n], i) = [x_0, \ldots, x_i + 1, \ldots, x_n]$$

et

$$\mathrm{dec}([x_0, \ldots, x_n], i) = [x_0, \ldots, \max(0, x_i - 1), \ldots, x_n]$$

Elles sont définies de la manière suivante :

$$\begin{aligned} \mathrm{inc}(l, 0) &= \mathrm{succ}(\mathrm{hd}(l)) :: \mathrm{tl}(l) \\ \mathrm{inc}(l, i+1) &= \mathrm{hd}(l) :: \mathrm{inc}(l, i) \end{aligned}$$

$$\begin{aligned} \mathrm{dec}(l, 0) &= \mathrm{pred}(\mathrm{hd}(l)) :: \mathrm{tl}(l) \\ \mathrm{dec}(l, i+1) &= \mathrm{hd}(l) :: \mathrm{dec}(l, i) \end{aligned}$$

On peut à présent définir tr_2 :

$$\begin{array}{lcll} \mathrm{tr}_2(e,p) & = & \mathrm{inc}(\pi_2(p), \pi_2(\mathrm{cur}(e,p))) & \text{si } \pi_1(\mathrm{cur}(e,p)) = 0 \quad /* \text{ incrémentation } */ \\ & = & \mathrm{dec}(\pi_2(p), \pi_2(\mathrm{cur}(e,p))) & \text{si } \pi_1(\mathrm{cur}(e,p)) = 1 \quad /* \text{ décrémentation } */ \\ & = & \pi_2(p) & \text{sinon} \quad /* \textit{autre} */ \end{array}$$

Il est claire que tr_2 est primitive récursive.

Fin de la preuve :

On défini à présent la fonction $\mathrm{tr} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ de transition d'un état à au autre :

$$\mathrm{tr}(e,p) = \langle \mathrm{tr}_1(e,p), \mathrm{tr}_2(e,p) \rangle$$

On défini à présent la fonction primitive récursive $\mathrm{st} : \mathbb{N} \times \mathbb{N}^n \times \mathbb{N} \to \mathbb{N}$ telle que $\mathrm{st}(e, x_1, \ldots, x_n, t)$ renvoie l'état de la machine qui exécute le programme e, avec les registres $R_1, \ldots, R_n$, initialisés respectivement à $x_1, \ldots, x_n$, après t étapes de calcul.

$$\begin{aligned} \mathrm{st}(e, x_1, \ldots, x_n, 0) &= \mathrm{Init}(e, x_1, \ldots, x_n) \\ \mathrm{st}(e, x_1, \ldots, x_n, t+1) &= \mathrm{tr}(e, \mathrm{st}(e, x_1, \ldots, x_n, t)) \end{aligned}$$

On arrive enfin à l'étape pour laquelle on a besoin de schéma μ, laissant la possibilité à une fonction de ne pas être définie sur certaine entrés. La fonction récursive $\mathrm{time} : \mathbb{N} \times \mathbb{N}^n \to \mathbb{N}$ donne le plus petit temps de calcul nécessaire pour que la machine arrive à l'instruction halt. La fonction sera définie si et seulement si la machine s'arrête pour le programme et les entrées correspondantes.

$$\mathrm{time}(e, x_1, \ldots, x_n) = \mu t. \pi_1(\mathrm{cur}(e, \mathrm{st}(e, x_1, \ldots, x_n, t))) = 4$$

Finalement, voici la fonction récursive qui correspond au calcul de la machine M. On lance la fonction de transition pour le nombre d'étapes nécessaires avant que la machine n'atteigne l'instruction halt, et on renvoie la valeur du registre R_0 :

$$f(e, x_1, \ldots, x_n) = \mathrm{hd}(\pi_2(\mathrm{st}(e, x_1, \ldots, x_n, \mathrm{time}(e, x_1, \ldots, x_n))))$$

Ceci conclut la démonstration. ∎

Corollaire 2.4.6 : Toute fonction calculable par un programme goto est aussi calculable par un programme structuré.

PREUVE: D'après le théorème 2.4.5, toute fonction calculable par un programme goto est une fonction récursive. D'après le théorème 2.3.4, toute fonction récursive est calculable par un programme structuré. ∎

La preuve du théorème 2.4.5 contient deux éléments que nous allons réutiliser dans la section 3.1 :

— La fonction cur : $\mathbb{N}\times\mathbb{N} \to \mathbb{N}$ qui à un code de programme e et à un état de la machine p, associe le code de l'instruction courante.

— La fonction st : $\mathbb{N}\times\mathbb{N}^n\times\mathbb{N} \to \mathbb{N}$ qui à un code de programme e, des valeurs $x_1, \ldots, x_n$ et un temps de calcul t, associe l'état de la machine de code e sur les entrées $x_1, \ldots, x_n$, après t étapes de calcul.

La fonction *st* sera en particulier très utile, car elle permet d'obtenir le résultat d'un calcul après t étape, *uniformément* en e, le code d'un programme : c'est à dire que c'est la même fonction pour tous les programmes, seul change le paramètre e qui donne le code du programme. Nous y reviendrons dans la section 3.1. Avant cela nous allons voir que les fonctions primitives récursives sont exactement les fonctions calculables par un programme structuré sans boucles `while`.

2.5 Programmes `for` et fonctions primitives récursives

Nous montrons dans cette section que les fonctions primitives récursives sont exactement les fonctions calculables par un programme structuré sans boucles `while`. Nous avons déjà vu une direction avec le théorème 2.3.4 : toute fonction primitive récursive est calculable par un programme structuré sans boucles `while`. Nous montrons à présent la réciproque :

Théorème 2.5.1:
Toute fonction calculable par un programme structuré sans boucles `while` *est primitive récursive.*

PREUVE: Pour un k donnée, et pour un programme structuré P sans boucles `while` et utilisant au plus les registres $R_0, \ldots, R_k$, on définie la fonction $f_P : \mathbb{N} \to \mathbb{N}$ par $f_p(\langle x_0, \ldots, x_k\rangle) = \langle v_0, \ldots, v_k\rangle$ où v_i est la valeur du registre R_i en fin d'exécution du programme P, quand son exécution commence avec ses registres initialisés aux valeurs $x_0, \ldots, x_k$.

Montrons que pour tout programme structuré P sans boucles `while`, la fonction f_P correspondante est primitive récursive. On montre le théorème par induction sur la hauteur des programmes structurés. On commence par montrer que c'est vrai pour les programmes structurés de hauteur 0. Pour le montrer on utilise une deuxième induction, cette fois-ci sur la taille des programmes structurés de hauteur 0.

On utilise pour cela les fonctions primitives récursives inc_i et dec_i définies par :

$$\begin{aligned} \mathrm{inc}_i(\langle x_0, \ldots, x_i, \ldots, x_k\rangle) &= \langle x_0, \ldots, x_i + 1, \ldots, x_k\rangle \\ \mathrm{dec}_i(\langle x_0, \ldots, x_i, \ldots, x_k\rangle) &= \langle x_0, \ldots, x_i - 1, \ldots, x_k\rangle \end{aligned}$$

Si P est un programme de taille 0 (ne contient aucune instruction, où alors seulement l'instruction halt), la fonction f_P est dans ce cas la fonction constante 0, qui est primitive récursive.

Supposons à présent par induction que la fonction f_P est primitive récursive, pour tous les programmes P de hauteur 0 et de longueur n. Montrons que c'est aussi le cas pour tous les programmes P de hauteur 0 et de longueur $n+1$. Soit P un tel programme. Alors P est le programme P' de longueur n, avec en plus une instruction $R_i = R_i+1$ où $R_j = R_j-1$. On a $f_p(\langle x_0,\ldots,x_k\rangle) = \mathrm{inc}_i(f_{p'}(\langle x_0,\ldots,x_k\rangle))$ ou bien $f_p(\langle x_0,\ldots,x_k\rangle) = \mathrm{dec}_j(f_{p'}(\langle x_0,\ldots,x_k\rangle))$.

Par induction la fonction f_P est donc primitive récursive pour tous les programmes P de hauteur 0. Nous faisons à présent l'induction sur la hauteur des programmes. Supposons que la fonction f_P existe pour tous les programmes P de hauteur n, et montrons qu'elle existe pour tous les programmes P de hauteur $n+1$.

1. Supposons d'abord que P a la forme suivante : if $R_j = 0$ then Q else Q' pour Q et Q' de hauteur au plus n. Par hypothèse d'induction les fonctions f_Q et f'_Q sont primitives récursives. La fonction f_P est donc donnée par :

$$\begin{array}{rcll} f_P(\langle x_0,\ldots,x_k\rangle) & = & f_Q(\langle x_0,\ldots,x_k\rangle) & \text{si } x_j = 0 \\ f_P(\langle x_0,\ldots,x_k\rangle) & = & f_{Q'}(\langle x_0,\ldots,x_k\rangle) & \text{sinon} \end{array}$$

2. Supposons ensuite que P à la forme suivante : for $i = 0$ to R_j do Q pour Q de hauteur au plus n. Par hypothèse d'induction la fonction f_Q est primitive récursive. La fonction f_P est donc donnée par :

$$f_P(\langle x_0,\ldots,x_k\rangle) = g(\langle x_0,\ldots,x_k\rangle, x_j)$$

où

$$\begin{array}{rcl} g(\langle x_0,\ldots,x_k\rangle, 0) & = & \langle x_0,\ldots,x_k\rangle \\ g(\langle x_0,\ldots,x_k\rangle, z+1) & = & f_Q(g(\langle x_0,\ldots,x_k\rangle, z)) \end{array}$$

3. A présent pour n'importe quel programme P de hauteur $n+1$ est une combinaison finie d'instructions $I_1,\ldots I_m$ de la forme (1) et (2) ou bien de la forme d'une incrémentation où d'une décrémentation. Une fonction primitive récursive f_{I_i} correspond à chacune de ces instructions. La fonction f_P est donnée par :

$$f_P = f_{I_m}(f_{I_{m-1}}(\ldots f_1(\langle x_0,\ldots,x_k\rangle))\ldots)$$

Par induction on a donc f_P primitive récursive, pour tout programme structuré P sans boucle `while`. A présent la fonction $f(x_1,\ldots,x_z)$ calculable par un programme structuré P sans boucles `while` utilisant au plus les registres $R_0,\ldots,R_k$ (pour $z \le k$) est donnée par $f(x_1,\ldots,x_z) = \pi_1(f_p(\langle 0, x_1,\ldots,x_z, 0\ldots,0\rangle))$. ∎

Chapitre 3

Fonctions universelles

3.1 La forme normale de Kleene

Nous utilisons ici les fonctions st et cur du théorème 2.4.5 afin de montrer un théorème qui nous permettra de travailler plus simplement avec les fonctions calculables.

Théorème 3.1.1:
Pour tout n, il existe un prédicat primitif récursif $T : \mathbb{N} \times \mathbb{N}^n \times \mathbb{N} \to \{0,1\}$ et une fonction primitive récursive $U : \mathbb{N} \to \mathbb{N}$ tel que pour toute fonction récursive $f : \mathbb{N}^n \to \mathbb{N}$, il existe un entier e tel que :

1. $\exists t\ T(e, x_1, \ldots, x_n, t) \leftrightarrow f(x_1, \ldots, x_n)\downarrow$
2. $\forall t\ T(e, x_1, \ldots, x_n, t) \to U(t) = f(x_1, \ldots, x_n)$

En particulier pour toute fonction récursives $f : \mathbb{N}^n \to \mathbb{N}$, on a un entier e tel que :

$$f(x_1, \ldots, x_n) = U(\mu t.T(e, x_1, \ldots, x_n, t))$$

Preuve: On reprend les fonctions *cur* et *st* de la preuve du théorème 2.4.5. Le prédicat T est informellement : la machine à registre atteind l'instruction halt après t étapes sur le programme codé par e avec les registres $R_1, \ldots, R_n$ initialisés à $x_1, \ldots, x_n$. Afin de définir la fonction U, on a besoin de garder les informations correspondant à $e, x_1, \ldots, x_n$. Pour cette raison, on encode ces informations dans le paramètre correspondant au temps de calcul. On définie alors $T(e, x_1, \ldots, x_n, t)$ comme le prédicat :

$$\pi_1(t) = \langle e, x_1, \ldots, x_n\rangle \wedge \pi_1(\mathrm{cur}(e, \mathrm{st}(e, x_1, \ldots, x_n, \pi_2(t)))) = 4$$

En particulier t doit être de la forme $\langle\langle e, x_1, \ldots, x_n\rangle, z\rangle$. On défini ensuite :

$$U(\langle\langle e, x_1, \ldots, x_n\rangle, z\rangle) = \mathrm{hd}(\pi_2(\mathrm{st}(e, x_1, \ldots, x_n, z)))$$

Fixons une fonction récursive f. D'après le théorème 2.3.4, il existe un entier e codant pour un programme goto calculant la fonction f. On a donc :

$$f(x_1, \ldots, x_n) = U(\mu t.T(e, x_1, \ldots, x_n, t))$$

pour tout $x_1, \ldots x_n$. ∎

Notation

On notera la fonction $\Phi_e : \mathbb{N}^n \to \mathbb{N}$ la fonction $U(\mu t.T(e, x_1, \ldots, x_n, t))$. Le théorème précédant nous dis la chose suivante : La fonction $\Phi(e, x_1, \ldots, x_n) = \Phi_e(x_1, \ldots, x_n)$ est récursive. On dira que $\{\Phi_e\}_{e\in\mathbb{N}}$ est une énumération calculable des fonctions récursives.

Notez que des entiers e ne correspondent pas à des codes de programme. Néanmoins la fonction partielle Φ_e est toujours bien définie même pour e qui n'est pas le code d'un programme : en effet pour $e, x_1, \ldots, x_n, t$ n'importe quels entiers, le prédicat $T(e, x_1, \ldots, x_n, t)$ est bien défini. Aussi si e ne correspond pas à un code de programme, pour certaines entrés $x_1, \ldots, x_n$, il existera peut-être t tel que $T(e, x_1, \ldots, x_n, t)$ est vrai (auquel cas $U(\mu t.T(e, x_1, \ldots, x_n, t)) \downarrow$ renverra une valeur), ou peut-être pas, auquel cas $U(\mu t.T(e, x_1, \ldots, x_n, t)) \uparrow$. Dans tous les cas même si e ne correspond pas à un code de programme, la fonction $x_1, \ldots, x_n \mapsto U(\mu t.T(e, x_1, \ldots, x_n, t))$ est récursive, et il y a donc un code de programme e' qui correspond à cette fonction.

Notation

Pour tout $n \in \mathbb{N}^*$, on considère qu'une énumération calculable $\{\Phi_e\}_{e\in\mathbb{N}}$ des fonctions récursives de $\mathbb{N}^n$ dans $\mathbb{N}$ est fixée une fois pour toute. C'est à dire qu'on a une fonction récursive partielle $\Phi(e, x_1, \ldots, x_n)$ telle que :

- Pour tout e, la fonction $x_1, \ldots, x_n \mapsto \Phi(e, x_1, \ldots, x_n)$ est récursive partiel,
- Pour toute fonction récursive partielle $f : \mathbb{N}^n \to \mathbb{N}$, il existe un e tel que $f(x_1, \ldots, x_n) = \Phi(e, x_1, \ldots, x_n)$.

Tous les théorèmes à venir sont indépendants du choix de cette énumération.

On définie à présent les prédicats primitifs récursifs $\Phi_e(n_1, \ldots, n_k)[t] \downarrow$ et $\Phi_e(n_1, \ldots, n_k)[t] \uparrow$

Notation

Étant donnée une fonction calculable $\Phi_e : \mathbb{N}^k \to \mathbb{N}$, on écriera $\Phi_e(n_1, \ldots, n_k)[t] \downarrow$ si informellement, la fonction Φ_e s'arrête en moins de t étapes de calcul. Formellement, avec $\Phi_e(x_1, \ldots, x_n) = U(\mu t\ T(e, x_1, \ldots, x_n, t))$, le prédicat $\Phi_e(n)[t] \downarrow$ est défini comme étant le prédicat : $\exists s < t\ T(e, x_1, \ldots, x_n, s)$. Le prédicat $\Phi_e(n_1, \ldots, n_k)[t] \uparrow$ est défini à l'inverse comme $\forall s < t\ \neg T(e, x_1, \ldots, x_n, s)$.

3.2 Le théorème SNM

La preuve du théorème SNM demande d'étendre nos manière de manipuler des listes avec des fonctions primitives récursives. Pour cela les deux exercices suivants vont nous aider.

Exercice 3.2.1

1. Montrez que les fonctions primitives récursives sont closes par récurrence primitive sur la suite des valeurs : si $g : \mathbb{N}^a \to \mathbb{N}$ et $h : \mathbb{N}^{a+2} \to \mathbb{N}$ sont primitives

récursives, alors la fonction $f : \mathbb{N}^{a+1} \to \mathbb{N}$ définit par :

$$\begin{array}{rcl} f(x_1,\ldots,x_a,0) & = & g(x_1,\ldots,x_a) \\ f(x_1,\ldots,x_a,n+1) & = & h(x_1,\ldots,x_a,n,[f(x_1,\ldots,x_a,n),\ldots,f(x_1,\ldots,x_a,0)]) \end{array}$$

est primitive récursive.

2. Montrez que si $g : \mathbb{N}^a \to \mathbb{N}$ et $h : \mathbb{N}^{a+2} \to \mathbb{N}$ sont primitives récursives, et si $d_1,\ldots,d_k : \mathbb{N} \to \mathbb{N}$ sont primitives récursives et telles que $d_i(n) \leq n$ pour tout n et $i \leq k$, alors la fonction $f : \mathbb{N}^{a+1} \to \mathbb{N}$ définit par :

$$\begin{array}{rcll} f(x_1,\ldots,x_a,0) & = & g(x_1,\ldots,x_a) & \\ f(x_1,\ldots,x_a,n+1) & = & h(x_1,\ldots,x_a,n, & f(x_1,\ldots,x_a,d_1(n)), \\ & & & \ldots, \\ & & & f(x_1,\ldots,x_a,d_k(n))) \end{array}$$

est primitive récursive.

Exercice 3.2.2

On défini la notation @ pour une fonction de $\mathbb{N}^2$ vers $\mathbb{N}$, telle que

$$[x_0,\ldots,x_n]@[y_0,\ldots,y_m] = [x_0,\ldots,x_n,y_0,\ldots,y_m]$$

Montrez que la fonction $@ : \mathbb{N}^2 \to \mathbb{N}$ est primitive récursive.

Le théorème suivant nous dit que l'on peut de manière uniforme, calculer à partir du code e d'une machine à $n+m$ paramètres, calculer le code de la machine qui fonctionne comme la machine de code e, après avoir fixer les n premiers paramètres :

Théorème 3.2.3:
Pour tout $n, m \in \mathbb{N}^*$ avec $m < n$, il existe une fonction primitive récursive $s_n^m : \mathbb{N}^{m+1} \to \mathbb{N}$ telle que pour tout e :

$$\Phi_{S_n^m(e,x_1,\ldots,x_m)}(y_1,\ldots,y_n) = \Phi_e(x_1,\ldots,x_m,y_1,\ldots,y_n)$$

PREUVE: Pour cette preuve, il est utile de revenir aux machines à registre. On utilisera le codage des programmes goto donnée dans la preuve de 2.4.5, ainsi que le codage primitif récursif des listes.

La fonction S_n^m, étant donné un code $e = \langle k, [I_1,\ldots,I_a]\rangle$ d'un programme, et des valeurs $x_1,\ldots,x_m$, va renvoyer le code $e' = \langle k+m, [I'_1,\ldots,I'_b]\rangle$ du même programme, mais avec :

1. Les registres $R_{k+1},\ldots,R_{k+m}$ initialisés aux valeurs $x_1,\ldots,x_m$ au début du programme.
2. Toutes les occurrences de R_i pour $1 \leq i \leq m$, remplacées par R_{i+k}
3. Toutes les occurrences de R_i pour $m < i$, remplacées par R_{i-m}
4. Toutes les occurrences de "goto l" remplacées par des occurrences de "goto l + p" où p est le nombre d'instructions à rajouter au début du programme pour initialiser les registres $R_{k+1},\ldots,R_{m+k}$ aux valeurs $x_1,\ldots,x_m$.

Il est clair qu'un tel programme fera l'affaire. Il faut juste montrer que l'on peut l'obtenir de façon primitive récursive. On commence par s'occuper de la fonction $g : \mathbb{N}^m \to \mathbb{N}$ qui détermine les instructions à rajouter au début du programme :

$$\begin{array}{rcll} g(0,\ldots,0) & = & [] & \text{le cas où } x_1,\ldots,x_m = 0 \\ g(0,\ldots,0,x_i,\ldots,x_m) & = & \langle 0,i\rangle :: g(0,\ldots,0,x_i-1,\ldots,x_m) & \text{le cas où } x_i \text{ est le premier} \\ & & & \text{paramètre différent de 0} \end{array}$$

Le nombre d'instructions rajoutées par la fonction $g(x_1,\ldots,x_m)$ est égal à $x_1+\cdots+x_m$. On définit aussi la fonction primitive récursive $\mathrm{nb} : \mathbb{N}^m \to \mathbb{N}$ qui calcule ce nombre d'instructions.

Pour simplifier l'écriture $x_1,\ldots,x_m$ sera noté $\bar{x}$ dans la fonction suivante. Il s'agit de la fonction primitive récursive $h : \mathbb{N}^{m+2} \to \mathbb{N}$ tel que $h(k,\bar{x},[I_1,\ldots,I_a])$ renvoie la liste d'instructions $[I'_1,\ldots,I'_a]$, qui est le résultat de l'application à chaque instructions de $[I_1,\ldots,I_a]$, des opérations (2) à (4) décrite ci-dessus :

$$\begin{array}{rcll} h(k,\bar{x},[]) & = & [] & \\ h(k,\bar{x},\langle 0,i\rangle :: l) & = & \langle 0,i+k\rangle :: h(k,\bar{x},l) & \text{si } 1 \le i \le m \\ & = & \langle 0,i-m\rangle :: h(k,\bar{x},l) & \text{si } i > m \\ & & & \\ h(k,\bar{x},\langle 1,i\rangle :: l) & = & \langle 1,i+k\rangle :: h(k,\bar{x},l) & \text{si } 1 \le i \le m \\ & = & \langle 1,i-m\rangle :: h(k,\bar{x},[I_2,\ldots,I_a]) & \text{si } i > m \\ & & & \\ h(k,\bar{x},\langle 2,n\rangle :: l) & = & \langle 2,\langle n+\mathrm{nb}(\bar{x})\rangle :: h(k,\bar{x},l) & \\ & & & \\ h(k,\bar{x},\langle 3,\langle i,n\rangle\rangle :: l) & = & \langle 3,\langle i+k,n+\mathrm{nb}(\bar{x})\rangle\rangle :: h(k,\bar{x},l) & \text{si } 1 \le i \le m \\ & = & \langle 3,\langle i-m,n+\mathrm{nb}(\bar{x})\rangle\rangle :: h(k,\bar{x},l) & \text{si } i > m \\ & & & \\ h(k,\bar{x},\langle 4,0\rangle :: l) & = & \langle 4,0\rangle\rangle :: h(k,\bar{x},l) & \end{array}$$

La fonction $S_n^m(\langle k,[I_1,\ldots,I_a]\rangle,x_1,\ldots,x_m)$ est donnée par la valeur

$$\langle k+m, g(x_1,\ldots,x_m)@h(k,x_1,\ldots,x_m,[I_1,\ldots,I_a])\rangle$$

La fonction h est primitive récursive en utilisant le deuxième schéma de l'exercice 3.2.1. Cela conclue la preuve. ∎

3.3 Le théorème du point fixe

Le théorème suivant nous dit que pour toute fonction qui modifie des programmes, il existe un programme dont le comportement n'est pas modifié par la fonction :

Théorème 3.3.1:
Pour toute fonction totale $f : \mathbb{N} \to \mathbb{N}$, il existe $e \in \mathbb{N}$ telle que $\forall n\ \Phi_{f(e)}(n) = \Phi_e(n)$.

Preuve: Soit a le code d'une machine à un paramètre, qui sur l'entré n, retourne le code d'une machine à un paramètre m, qui :

1. calcul la valeur $f(\Phi_n(n))$
2. si on a $f(\Phi_n(n))\downarrow$, alors retourne le résultat du calcul de la machine de code $f(\Phi_n(n))$ sur l'entré m (et sinon ne s'arrête pas).

Formellement a est tel que pour tout $n, m \in \mathbb{N}$ on a :

$$\Phi_{\Phi_a(n)}(m) = \Phi_{f(\Phi_n(n))}(m)$$

Notez que Φ_a est une fonction totale : pour tout n, dans le calcul de $\Phi_a(n)$, on ne cherche pas à faire les étapes (1) et (2), mais seulement à calculer le code d'une machine qui fait les étapes (1) et (2). La démonstration qu'un tel code a existe est donné par le théorème SNM : La fonction $\Phi_{f(\Phi_n(n))}(m)$ est récursive. Il y a donc un code b tel que $\Phi_b(n, m) = \Phi_{f(\Phi_n(n))}(m)$. D'après le théorème SNM, il y a une fonction primitive récursive s telle que $\Phi_{s(b,n)}(m) = \Phi_{f(\Phi_n(n))}(m)$. Comme s est primitif récursif, il y a un code a tel que $\Phi_a(n) = s(b, n)$.

Le point fixe sera alors $\Phi_a(a)$. En effet on a :

$$\forall m\ \Phi_{\Phi_a(a)}(m) = \Phi_{f(\Phi_a(a))}(m)$$

Ceci conclue la preuve. ∎

En pratique, le théorème du point fixe est souvent utilisé sous la forme suivante : un programme informatique peut toujours avoir accès à son propre code. En effet, supposons qu'un programme M utilise une variable `var` ayant été initialisé à une certaine valeur au début de son exécution. On peut alors facilement définir une fonction primitive récursive f qui prend n en paramètre, et retourne le code de M, qui commence avec `var` initialisée à n. D'après le théorème du point fixe, il y a une valeur e telle que les machines de code e et $f(e)$ ont le même comportement. Notez alors que pour tout e, la fonction $f(e)$ renvoie toujours le code de M avec pour seule différence que `var` est initialisé à e. En particulier le programme de code $f(e)$ aura le code e dans la variable `var`. Aussi ce code $f(e)$ est équivalent au code e. Donc e code pour le programme M avec la variable `var` initialisé à e : il y a une version de M qui peut accéder à son propre code.

Voici un exemple en C qui illustre ce que dit le théorème du point fixe :

Exemple 3.3.2 : Dans l'exemple suivant, la variable `res` contient une chaîne de caractère qui est égale au programme lui-même : La fonction `sprintf` va recopier la chaîne de caractère `prg` dans `res`, en remplaçant le "%s" par `prg` lui-même. Les "%c" sont là uniquement pour gérer les problèmes techniques de retour à la ligne et de guillemets.

```
#include <stdio.h>
int main() {
char* prg="#include <stdio.h>%cint main() {%cchar* prg=%c%s%c;%cchar
    res[1000];%csprintf(res, prg, 10, 10, 34, prg, 34, 10, 10, 10, 10)
    ;%c}";
char res[1000];
sprintf(res, prg, 10, 10, 34, prg, 34, 10, 10, 10, 10);
}
```

L'exemple précédent peut s'étendre à n'importe quel programme : après la dernière instruction, on peut rajouter toutes les instructions que l'on veut, y compris des instructions qui utilisent le contenu de `res`. Afin que `res` contienne aussi ces nouvelles instructions, il faut simplement aussi les rajouter à la fin de la chaine de caractère `prg`.

Exercice 3.3.3

Soit $A \subseteq \mathbb{N}$ tel que $A \neq \mathbb{N}$ et A est calculable.

1. Montrez qu'il existe une fonction récursive f telle que $x \in A$ iff $f(x) \notin A$.
2. En utilisant le théorème du point fixe, en déduire qu'il existe i, j tel que $\Phi_i = \Phi_j$ avec $i \in A$ et $j \notin A$.
3. En déduire qu'il n'existe pas de prédicats calculables P tel que $P(e)$ ssi e est le code d'un programme qui fait la multiplication par 2.
4. Généralisez la question précédente pour montrer le théorème de Rice : Pour tout comportement non-trivial des programmes, il n'est pas possible de calculer l'ensemble des codes de programmes ayant ce comportement. Formellement : soit un ensemble $C \neq \mathbb{N}$ de codes de fonctions tel que pour tout e_1, e_2 avec $\phi_{e_1} = \phi_{e_2}$, on a $e_1 \in C \to e_2 \in C$. Alors C n'est pas calculable.

3.4 La boucle `while` est indispensable

Nous montrons dans cette section que la boucle `while` est indispensable dans les programmes structurés : il existe des fonctions totales calculable par des programmes structurés, qui ne sont pas calculables par des programmes structurés sans boucle `while`. D'après les théorèmes 2.3.4 et 2.5.1, les fonctions primitives récursives sont exactement les fonctions calculables par des programmes sans boucle `while`. Il suffit donc simplement de montrer qu'il existe des fonctions récursives totales, qui ne sont pas primitives récursives.

Intuitivement, un argument de diagonalisation doit fonctionner : Il est possible d'obtenir une énumération des fonctions récursives $\{\Phi_e\}_{e\in\mathbb{N}}$ et un prédicat récursif $P \subseteq \mathbb{N}$ tel que $P(e)$ implique que e code pour une fonction primitive récursive, et tel que toute fonction primitive récursive a un code e pour lequel on a $P(e)$. A partir de là on définit la fonction récursive f qui énumère les codes des fonctions primitives récursives : $f(0) = \min\{e : P(e)\}$ et $f(n+1) = \min\{e : P(e) \text{ et } e \neq f(0), \ldots, f(n)\}$. On définit ensuite facilement une fonction récursive totale différente de toutes les fonctions primitives récursives de la manière suivante :

$$\begin{array}{rcll} g(n) & = & 1 & \text{si } \Phi_{f(n)}(n) = 0 \\ g(n) & = & 0 & \text{sinon} \end{array}$$

La démonstration de l'existence du prédicat P serait fastidieuse. Aussi nous allons procéder autrement, en utilisant la fonction d'Ackermann, le premier exemple historique de fonction récursive qui n'est pas primitive récursive :

Définition 3.4.1 : On définit les fonctions $A_n : \mathbb{N} \to \mathbb{N}$ par induction sur $n \in \mathbb{N}$ de la manière suivante :

— A_0 est la fonction $x \mapsto 2^x$

— $A_{n+1}(x)$ est l'application x fois de la fonction A_n sur 1 : $A_n(A_n(\ldots(A_n(1))))$. ◊

Formellement :

$$\begin{array}{rcl} A_0(x) & = & 2^x \\ A_{n+1}(0) & = & 1 \\ A_{n+1}(x) & = & A_n(A_{n+1}(x-1)) \end{array}$$

Exercice 3.4.2

Montrez que pour tout n, la fonction A_n est primitive récursive.

Définition 3.4.3 : La fonction d'Ackermann est la fonction $n \mapsto A_n(n)$. ◊

La fonction d'Ackermann a une croissance extrêmement rapide. Ainsi $A(0) = 1$, $A(1) = 2$, $A(2) = 16$, et $A(3)$ est déjà égal à 65536 itérations de la fonction $x \mapsto 2^x$ (en commençant sur 0), c'est à dire :

$$A(3) = 2^{\left(2^{\left(\cdots^{(2^0)}\right)}\right)} \text{ où la puissance est itérée 65536 fois}$$

Malgré sa très forte croissance, intuitivement la fonction d'Ackermann devrait être calculable : Pour calculer $A_n(n)$, on peut utiliser une pile contenant soit des fonctions A_n (en pratique une représentation de ces fonctions), soit des entiers. Par exemple si on empile A_n, A_{n+1} et ensuite k, cela correspond au calcul $A_n(A_{n+1}(k))$. Ainsi le sommet de la pile est toujours un entier, et l'élément qui suit (si il existe) est toujours une fonction. Aussi pour calculer $A_n(n)$ on procède comme suit :

1. On empile A_n, puis on empile n.
2. Tant que la pile contient plus d'un élément :
   (a) On dépile l'entier k, puis on dépile la fonction A_m.
   (b) Si $m = 0$ on empile 2^k.
   (c) Sinon, si $k = 0$ on empile 1.
   (d) Sinon on empile A_{m-1}, puis A_m et enfin $k - 1$.

L'algorithme s'arrêtera quand la pile ne contiendra plus qu'un élément, le résultat du calcul $A_n(n)$.

Nous allons donner à présent une preuve plus formel et sans doute un peu plus mystérieuse, du fait que la fonction d'Ackermann soit récursive, en utilisant le théorème du point fixe.

Théorème 3.4.4:
La fonction d'Ackermann est récursive.

PREUVE: On définit la fonction récursive $g : \mathbb{N}^3 \to \mathbb{N}$ suivante :

$$\begin{array}{rcll} g(e,n,m) & = & 2^m & \text{si } n = 0 \\ & = & 1 & \text{si } m = 0 \\ & = & \Phi_e(n-1, \Phi_e(n, m-1)) & \text{sinon} \end{array}$$

Comme g est récursive, il existe un entier a tel que $g(e,n,m) = \Phi_a(e,n,m)$. En utilisant le théorème SNM, on a une fonction primitive récursive $f : \mathbb{N} \to \mathbb{N}$ telle que $\Phi_{f(e)}(n,m) = \Phi_a(e,n,m)$. A présent, d'après le théorème du point fixe, il existe un code k tel que $\Phi_{f(k)}(n,m) = \Phi_k(n,m)$. En reprenant la définition de $\Phi_a(e,n,m)$, et en utilisant $\Phi_{f(k)} = \Phi_k$ on a donc :

$$\begin{array}{rcll} \Phi_k(n,m) & = & 2^m & \text{si } n = 0 \\ & = & 1 & \text{si } m = 0 \\ & = & \Phi_k(n-1, \Phi_k(n, m-1)) & \text{sinon} \end{array}$$

La fonction $n \mapsto \Phi_k(n,n)$ est donc la fonction d'Ackermann, et elle est récursive puisqu'elle a k pour code. ■

Nous allons maintenant montrer que la fonction d'Ackermann n'est pas primitive récursive. Nous allons montrer pour cela que la fonction $n \mapsto A_n(n)$ croît plus vite que toutes les fonctions primitives récursives, c'est à dire que pour une fonction primitive récursive f, il existe k tel que pour tout $n \geq k$ on a $A_n(n) > f(n)$. On a pour cela besoin de plusieurs résultats intermédiaires :

Lemme 3.4.5 : Pour tout n, on a $A_n(x) > x$. ⋆

PREUVE: On a $A_0(x) = 2^x > x$, donc le lemme est vrai pour A_0. Fixons n et supposons à présent le lemme vrai pour A_n. Montrons que le lemme est vrai pour A_{n+1}. On procède pour cela par récurrence sur x. On a $A_{n+1}(0) = 1 > 0$. Supposons à présent que pour x on ait $A_{n+1}(x) > x$. On a alors $A_{n+1}(x+1) = A_n(A_{n+1}(x))$. Comme le lemme est vrai pour n on a $A_n(A_{n+1}(x)) > A_{n+1}(x) \geq A_{n+1}(x) + 1$. Aussi par hypothèse de récurrence on a $A_{n+1}(x) > x$. On en déduit donc $A_{n+1}(x+1) > x+1$. ■

Lemme 3.4.6 : Pour tout n, la fonction A_n est croissante.
Pour tout x, la fonction $n \mapsto A_n(x)$ est croissante. ⋆

PREUVE: La fonction A_0 est clairement croissante. Ensuite pour tout n, on a $A_{n+1}(x+1) = A_n(A_{n+1}(x))$. En utilisant le lemme 3.4.5 on a $A_n(A_{n+1}(x)) > A_{n+1}(x)$ et donc $A_{n+1}(x+1) > A_{n+1}(x)$. Donc pour tout n la fonction A_{n+1} est croissante.

Montrons à présent que pour tout x et pour tout n, on a $A_{n+1}(x) \geq A_n(x)$. Pour $x = 0$, pour tout n on a $A_{n+1}(0) = 1$ et $A_n(0) = 1$. Donc $A_{n+1}(0) \geq A_n(0)$. Pour $x > 0$ on a $A_{n+1}(x) = A_n(A_{n+1}(x-1))$. D'après le lemme 3.4.5 on a $A_{n+1}(x-1) > x-1 \geq x$. Aussi comme la fonction A_n est croissante on a $A_n(A_{n+1}(x-1)) \geq A_n(x)$. On a donc $A_{n+1}(x) \geq A_n(x)$, ce qui implique que la fonction $n \mapsto A_n(x)$ est croissante. ■

On aborde à présent la notion clef pour la preuve. On utilise pour cela la notion de domination presque partout.

Notation

Un prédicat $P \subseteq \mathbb{N}$ est vrai pour *presque tous* les entiers x, si $P(x)$ est vrai pour pour tous les entiers x, sauf un nombre fini d'entre eux. On notera alors $\forall^* x\ P(x)$, ce qui doit être compris comme : $\exists y\ \forall x > y\ P(x)$.

La notation précédente sera aussi utilisé pour des prédicats $P \subseteq \mathbb{N}^n$ pour $n > 0$. Dans ce cas $\forall^* x_1, \ldots, x_n\ P(x_1, \ldots, x_n)$, doit être compris comme : $\exists y\ \forall x_1, \ldots, x_n > y\ P(x_1, \ldots, x_n)$. De manière équivalente $P(x_1, \ldots, x_n)$ est vrai pour tous les n-uplets $(x_1, \ldots, x_n)$, sauf un nombre fini d'entre eux.

Notation

Pour deux fonctions $f, g : \mathbb{N} \to \mathbb{N}$, on écriera $f <^* g$ si g domine f presque partout, c'est à dire $\forall^* x\ f(x) < g(x)$.

On introduit enfin un dernier concept :

Notation

Pour une fonction $f : \mathbb{N} \to \mathbb{N}$, on écriera $f^n(x)$ pour dénoter le résultat de l'application de f à x, n fois. C'est à dire $f^0(x) = x$ et $f^{n+1}(x) = f(f^n(x))$.

Le lemme suivant est clef pour montrer que la fonctions d'Ackermann domine toutes les fonctions primitives récursives presque partout. Il permettra en particulier de traiter du schéma de récurrence des fonctions primitives récursives.

Lemme 3.4.7 : Pour tout n, il existe $m > n$ tel que :

$$\forall^* x \; A_m(x) \geq A_n^{x+1}(x+1)$$

PREUVE: Montrons d'abord que pour tout n, x, y on a :

$$A_{n+1}(x+y) \geq A_n^y(x) \qquad (*)$$

La démonstration se fait par récurrence sur y. Pour $y = 0$ on a $A_{n+1}(x) \geq x$ d'après le lemme 3.4.5. Supposons que (*) soit vrai pour y, montrons que (*) est vrai pour $y+1$. On a $A_{n+1}(x+y+1) = A_n(A_{n+1}(x+y))$. Par l'hypothèse de récurrence on a $A_{n+1}(x+y) \geq A_n^y(x)$. On a donc $A_{n+1}(x+y+1) \geq A_n(A_n^y(x)) = A_n^{y+1}(x)$ (car A_{n+1} est croissante). On en déduit que (*) est vrai pour tout n, x, y.

En particulier on a $A_{n+1}(x+x+1) \geq A_n^{x+1}(x+1)$. Montrons à présent que pour $x \geq 5$, on a $A_{n+2}(x) \geq A_{n+1}(2x+1)$. Pour tout x on a $A_{n+2}(x) = A_{n+1}(A_{n+2}(x-1))$. Montrons que pour $x \geq 5$ on a $A_{n+2}(x-1) \geq 2x+1$. On a clairement que $2^{x-1} \geq 2x+1$ pour $x \geq 5$. Comme pour tout x on a $A_{n+2}(x) \geq A_0(x)$, on a aussi que $A_{n+2}(x-1) \geq 2x+1$ pour $x \geq 5$. On en déduit alors que $A_{n+1}(A_{n+2}(x-1)) \geq A_{n+1}(2x+1)$ pour $x \geq 5$ (car A_{n+1} est croissante), et donc que pour presque tout x, on a $A_{n+2}(x) \geq A_{n+1}(2x+1)$.

Comme $A_{n+1}(2x+1) \geq A_n^{x+1}(x+1)$ pour tout x, on en déduit alors que $A_{n+2}(x) \geq A_n^{x+1}(x+1)$ pour presque tout x, ce qui conclut la preuve. ∎

On est à présent en mesure de montrer que la fonction d'Ackermann domine presque partout toutes les fonctions primitives récursives.

Théorème 3.4.8:
La fonction d'Ackermann $A(n) = A_n(n)$ domine presque partout toutes les fonctions primitives récursives, et n'est donc pas primitive récursive.

PREUVE: Soit $n > 0$ et $f : \mathbb{N}^n \to \mathbb{N}$. On note $P(f)$ le prédicat :

$$\exists k \; \forall^* x_1, \ldots, x_n \; A_k(\max(x_1, \ldots, x_n)) \geq f(x_1, \ldots, x_n)$$

On va montrer que pour tous n et toute fonction primitive récursive $f : \mathbb{N}^n \to \mathbb{N}$, le prédicat $P(f)$ est vrai. Cela se montre par induction sur la hauteur des fonctions primitives récursives. $P(f)$ est clairement vérifié pour les fonctions primitives récursives f de hauteur 0, c'est à dire les fonctions constantes, les projections et la fonction successeur : elles sont toutes dominés presque partout par A_0.

Supposons à présent que l'on ait $P(h)$ et $P(g_1), \ldots P(g_n)$ pour les fonctions primitives récursives $h : \mathbb{N}^n \to \mathbb{N}$ et $g_1, \ldots, g_n : \mathbb{N}^m \to \mathbb{N}$. Montrons $P(f)$ pour la fonction $f(x_1, \ldots, x_m) = h(g_1(x_1, \ldots, x_m), \ldots, g_n(x_1, \ldots, x_m))$.

En particulier il existe k tel que $\forall^* x_1, \ldots, x_n$ et $\forall^* x_1, \ldots, x_m$:

$$\begin{aligned} h(x_1, \ldots, x_n) &\leq A_k(\max(x_1, \ldots, x_n)) \\ g_1(x_1, \ldots, x_m) &\leq A_k(\max(x_1, \ldots, x_m)) \\ &\ldots \\ g_n(x_1, \ldots, x_m) &\leq A_k(\max(x_1, \ldots, x_m)) \end{aligned}$$

On a donc, en utilisant le fait que A_k soit croissante :

$$\begin{aligned} h(g_1(x_1, \ldots, x_m), \ldots, g_n(x_1, \ldots, x_m)) &\leq A_k(\max(g_1(x_1, \ldots, x_m), \ldots, g_n(x_1, \ldots, x_m))) \\ &\leq A_k(A_k(\max(x_1, \ldots, x_m))) \\ &\leq A_k^2(\max(x_1, \ldots, x_m)) \end{aligned}$$

D'après le lemme 3.4.7, il existe t tel que $\forall^* x\ A_t(x) \geq A_n^{x+1}(x+1) \geq A_n^x(x)$. En particulier $\forall^* x\ A_t(x) \geq A_n^2(x)$ (on utilise ici $A_n(x) > x$). On a donc $\forall^* x_1, \ldots, x_m\ A_t(\max(x_1, \ldots, x_m)) \geq h(g_1(x_1, \ldots, x_m), \ldots, g_n(x_1, \ldots, x_m))$. Donc $P(f)$ est vérifié.

Supposons à présent que $P(h)$ et $P(g)$ soit vrai pour les fonctions primitives récursives $h : \mathbb{N}^{n+2} \to \mathbb{N}$ et $g : \mathbb{N}^n \to \mathbb{N}$. Montrons que $P(f)$ est vrai pour la fonction :

$$\begin{aligned} f(x_1, \ldots, x_n, 0) &= g(x_1, \ldots, x_n) \\ f(x_1, \ldots, x_n, x+1) &= h(x_1, \ldots, x_n, x, f(x_1, \ldots, x_n, x)) \end{aligned}$$

En particulier il existe k tel que $\forall^* x_1, \ldots, x_n, z, x$:

$$\begin{aligned} g(x_1, \ldots, x_n) &\leq A_k(\max(x_1, \ldots, x_n)) \\ h(x_1, \ldots, x_n, z, x) &\leq A_k(\max(x_1, \ldots, x_n, z, x)) \end{aligned}$$

Montrons par récurrence sur x que pour tout x on a :

$$f(x_1, \ldots, x_n, x) \leq A_k^{x+1}(\max(x_1, \ldots, x_n)) \qquad (*)$$

On a clairement $f(x_1, \ldots, x_n, 0) = g(x_1, \ldots, x_n) \leq A_k(\max(x_1, \ldots, x_n))$. Supposons à présent que (*) est vrai pour x et montrons que c'est vrai pour $x + 1$. On a

$$\begin{aligned} f(x_1, \ldots, x_n, x+1) &= h(x_1, \ldots, x_n, x, f(x_1, \ldots, x_n, x)) \\ &\leq A_k(\max(x_1, \ldots, x_n, x, A_k^{x+1}(\max(x_1, \ldots, x_n)))) \qquad (1) \\ &\leq A_k(A_k^{x+1}(\max(x_1, \ldots, x_n, x))) \qquad (2) \\ &\leq A_k^{x+2}(\max(x_1, \ldots, x_n, x+1)) \end{aligned}$$

Le passage de (1) à (2) utilise le fait que $A_k(x) > x$. On a donc que (*) est vérifié pour $x + 1$, et donc par récurrence pour tout x, c'est à dire $f(x_1, \ldots, x_n, x) \leq A_k^{x+1}(\max(x_1, \ldots, x_n))$. On a aussi clairement que :

$$A_k^{x+1}(\max(x_1, \ldots, x_n)) \leq A_k^{\max(x_1, \ldots, x_n, x+1)}(\max(x_1, \ldots, x_n, x+1))$$

A présent en utilisant le lemme 3.4.7, il existe t tel que $\forall^* x\ A_k^{x+1}(x+1) \leq A_t(x)$. On a donc

$$\forall^* x\ A_k^{\max(x_1, \ldots, x_n, x+1)}(\max(x_1, \ldots, x_n, x+1)) \leq A_t(\max(x_1, \ldots, x_n, x))$$

On en déduit $\forall^* x\ f(x_1, \ldots, x_n, x) \leq A_t(\max(x_1, \ldots, x_n, x))$ et donc $P(f)$.

Par récurrence on a donc $P(f)$ pour toute fonction primitive récursive f. Montrons enfin que la fonction d'Arckermann domine presque partout toutes les fonctions récursives. Pour toute fonction primitive récursive $f : \mathbb{N} \to \mathbb{N}$ il existe t tel que $\forall^* n\ f(n) + 1 \leq A_t(n)$. Aussi pour $n \geq t$ on a $A_t(n) \leq A_n(n)$. Donc la fonction d'Arckermann domine f presque partout, et c'est le cas pour toutes les fonctions primitives récursives. On en déduit enfin que la fonction d'Arckermann n'est pas primitive récursive. ∎

Chapitre 4

Premiers ensembles incalculables

4.1 Ensembles récursivement énumérable

On rappelle qu'un ensemble est calculable si jamais sa fonction caractéristique est calculable, formellement :

Définition 4.1.1 : Un ensemble $A \subseteq \mathbb{N}$ est calculable si il existe un entier e tel que $n \in A \leftrightarrow \Phi_e(n)\downarrow = 1$ et $n \notin A \leftrightarrow \Phi_e(n)\downarrow = 0$. ♢

On introduit maintenant un concept central : les ensembles d'entiers que l'on peut énumérer avec un programme informatique.

Définition 4.1.2 : Un ensemble $A \subseteq \mathbb{N}$ est récursivement énumérable si il existe un entier e tel que $n \in A \leftrightarrow \Phi_e(n)\downarrow$. ♢

Exercice 4.1.3

Montrez que tout ensemble calculable est récursivement énumérable.

On voit la machine de code e comme processus qui énumère A : Pour tous les entiers, on cherche un temps t tel que $\Phi_e(n)$ s'arrête en t étapes de calcul. Si on trouve un tel t, alors on énumère n dans notre ensemble. Evidemment on ne peut pas chercher le temps de calcul t pour tous les entiers en même temps, car il y en a une infinité. L'idée est de décomposer comme suit :

1. On regarde si $\Phi(0)[0]\downarrow$
2. On regarde si $\Phi(0)[1]\downarrow$ ou $\Phi(1)[1]\downarrow$
3. On regarde si $\Phi(0)[2]\downarrow$ ou $\Phi(1)[2]\downarrow$ ou $\Phi(2)[2]\downarrow$
4. ...

Cette idée est reprise dans la preuve de la proposition suivante :

Proposition 4.1.4 : Un ensemble infini A est récursivement énumérable ssi il existe une fonction calculable total $f : \mathbb{N} \to \mathbb{N}$ tel que $f(\mathbb{N}) = A$. ⋆

Preuve: Soit A un ensemble récursivement énumérable infini. Soit e tel que $\Phi_e(n)\downarrow$ ssi $n \in A$. Soit a le plus petit élément de A. On définit $g : \mathbb{N} \to \mathbb{N}$ comme suit :

$$\begin{aligned} g(0) &= \langle a, 0\rangle \\ g(n+1) &= \mu x.\big(\Phi_e(\pi_1(x))[\pi_2(x)]\downarrow \ \wedge\ \forall k < n+1\ \pi_1(x) \neq \pi_1(g(k))\big) \end{aligned}$$

Et on définie ensuite :

$$f(n) = \pi_1(g(n))$$

Il est claire que $f(\mathbb{N}) = A$. Supposons à présent que $f(\mathbb{N}) = A$ pour une fonction calculable totale A. Alors la fonction g définie par :

$$g(n) = \mu t.f(t) = n$$

est calculable et est telle que $g(n)\downarrow$ ssi $\exists t\ f(t) = n$. ∎

Nous allons voir qu'un ensemble récursivement énumérable n'est pas forcément calculable. La proposition suivante donne plus précisément les connexions entre ces deux concepts :

Proposition 4.1.5 : Un ensemble A est calculable ssi A et $\mathbb{N} - A$ sont tous les deux récursivement énumérables. ⋆

Preuve: Soit A un ensemble calculable. D'après l'exercice 4.1.3 il est récursivement énumérable. Par ailleurs l'ensemble $\mathbb{N} - A$ est par lui aussi calculable et par conséquent aussi récursivement énumérable.

Supposons à présent que l'ont ait deux codes e_1, e_2 tel que $\Phi_{e_1}(n)\downarrow$ ssi $n \in A$ et $\Phi_{e_2}(n)\downarrow$ ssi $n \in \mathbb{N} - A$. On définie la fonction calculable $f(n) = \mu t.(\Phi_{e_1,t}(n)\downarrow \text{ ou } \Phi_{e_2,t}(n)\downarrow)$. Notez que la fonction f est nécéssairement totale. On définie ensuite le code de fonction e tel que :

$$\begin{array}{lcll} \Phi_e(n) & = & 1 & \text{si } \Phi_{e_1}(n)[f(n)]\downarrow \\ \Phi_e(n) & = & 0 & \text{sinon} \end{array}$$

Il est claire que la fonction Φ_e calcul l'ensemble A. ∎

Comme nous l'avons dit, il existe des choses récursivement énumérables qui ne sont pas calculable. L'exemple canonique est connu comme étant "l'arrêt des programmes informatique" :

Définition 4.1.6 : On définie l'arrêt (noté $\emptyset'$)comme l'ensemble :

$$\emptyset' = \{n \in \mathbb{N} \ :\ \Phi_n(n)\downarrow\}$$

Proposition 4.1.7 : L'arrêt est un ensemble récursivement énumérable qui n'est pas calculable. ⋆

Preuve: La fonction partielle $f : n \mapsto \Phi_n(n)$ est clairement calculable, et on a $f(n)\downarrow$ ssi $\Phi_n(n)\downarrow$. Aussi par définition on a $\emptyset' = \{n \mid f(n)\downarrow\}$. Donc $\emptyset'$ est récursivement énumérable.

Supposons maintenant que $\emptyset'$ est un ensemble calculable. En particulier d'après le proposition 4.1.5, l'ensemble $\mathbb{N} - \emptyset'$ est récursivement énumérable, et il existe un code e tel que $n \in \mathbb{N} - \emptyset'$ ssi $\Phi_e(n)\downarrow$. On a alors pour tout n :

$$\Phi_e(n)\downarrow \leftrightarrow n \in \mathbb{N} - \emptyset' \leftrightarrow \Phi_n(n)\uparrow$$

En particulier pour $n = e$ on a

$$\Phi_e(e)\downarrow \leftrightarrow e \in \mathbb{N} - \emptyset' \leftrightarrow \Phi_e(e)\uparrow$$

Ce qui est une contradiction. Donc l'arrêt n'est pas calculable, et en particulier le complémentaire de l'arrêt n'est pas récursivement énumérable. ∎

Exercice 4.1.8

Un ensemble infini est récursivement énumérable dans l'ordre si il existe une fonction totale $f : \mathbb{N} \to \mathbb{N}$ telle que $f(\mathbb{N}) = A$ et telle que $f(n) < f(n+1)$. Montrez que si un ensemble infini A est récursivement énumérable dans l'ordre, alors A est calculable.

Exercice 4.1.9

Montrez que tout ensemble récursivement énumérable infini contient un sous-ensemble calculable infini.

4.2 Réduction many-one

L'arrêt est donc un exemple concret d'ensemble non-calculable. Aussi pour montrer en général qu'un ensemble est non calculable, il suffit de montrer qu'il est suffisement puissant pour calculer l'arrêt. On introduit pour cela la notion de réduction many-one :

Définition 4.2.1 : Soit deux ensemble A, B. On dit que A est many-one réductible à B, et on écrit $A \leq_m B$ si il existe une fonction récursive totale $f : \mathbb{N} \to \mathbb{N}$ telle que $n \in A \leftrightarrow f(n) \in B$. Si $A \leq_e B$ et $B \leq_e A$, on écrit $A \equiv_e B$. ♢

Quand on a $A \leq_m B$, la connaissance de B est suffisante pour calculer A : il existe une fonction totale calculable f tel que $n \in A \leftrightarrow f(n) \in B$. Ainsi pour savoir si $n \in A$, on utilise la fonction f pour "poser une question" à l'ensemble B : est-ce que $f(n) \in B$? si la réponse est oui, alors $n \in A$. Sinon $n \notin A$.

Proposition 4.2.2 : Supposons A non calculable, et supposons $A \leq_m B$. Alors B est non-calculable. ⋆

Preuve: Soit f calculable totale telle que $n \in A \leftrightarrow f(n) \in B$. Supposons B calculable par la fonction h. Alors A est calculable avec la fonction $g(n) = h(f(n))$. ■

Exercice 4.2.3

Montrez que l'ensemble des codes e tel que $\Phi_e(0) = 0$ n'est pas calculable.

Exercice 4.2.4

Montrez que l'ensemble des codes e tel que $\Phi_e(0) \downarrow \neq \Phi_e(1) \downarrow$ n'est pas calculable.

Exercice 4.2.5

Montrez que l'ensemble des codes e tel que $\forall n\ \Phi_e(n) \downarrow$ n'est pas calculable.

Exercice 4.2.6

Soit $A = \{e \ : \ \exists n \ \Phi_e(n)\uparrow\}$. Soit $B = \{e \ : \ \exists n \text{ pair } \ \Phi_e(n)\uparrow\}$. Montrez que $A \equiv_m B$.

A l'inverse, si $A \leq_e B$ et B est récursivement énumérable, alors nécéssairement on aura que A est récursivement énumérable.

Proposition 4.2.7 : Supposons B récursivement énumérable, et supposons $A \leq_m B$. Alors A est récursivement énumérable. ⋆

PREUVE: Soit f calculable totale telle que $n \in A \leftrightarrow f(n) \in B$. Supposons $n \in B$ ssi $\Phi_e(n)\downarrow$. Soit a le code du programme tel que $\Phi_a(n) = \Phi_e(f(n))$. On a en particulier $n \in A$ ssi $\Phi_e(f(n))\downarrow$ ssi $\Phi_a(n)\downarrow$. ■

Nous montrons à présent que l'arrêt est *complet* pour les ensembles récursivement énumérables. C'est à dire que l'arrêt est récursivement énumérable et que tous les ensembles récursivement énumérables se réduisent à lui.

Proposition 4.2.8 : Un ensemble A est récursivement énumérable ssi $A \leq_m \emptyset'$. ⋆

PREUVE: La proposition 4.2.7 implique que si $A \leq_m \emptyset'$ alors A est récursivement énumérable. Montrons à présent que si A est récursivement énumérable, alors $A \leq_m \emptyset'$.

Supposons que A soit récursivement énumérable. Soit e tel que $n \in A$ ssi $\Phi_e(n)\downarrow$. Soit f la fonction primitive récursive qui à n, retourne le code du programme tel que pour tout m on ait $\Phi_{f(n)}(m) = \mu t.\Phi_e(m)[t]\downarrow$. En particulier on a $n \in A$ ssi $\Phi_e(n)\downarrow$ ssi $\Phi_{f(n)}(f(n))\downarrow$ ssi $f(n) \in \emptyset'$. On a donc $A \leq_m \emptyset'$. ■

4.3 Plus d'ensembles non-calculables

Les exercices 4.2.3 à 4.2.6 ont pour but de montrer qu'une série d'ensembles ne sont pas calculables. On utilise pour cela toujours la même technique : pour ces exercices, on montre qu'un ensemble A n'est pas calculable car on a $\emptyset' \leq_e A$. On pose ici une première question : Si un ensemble est non-calculable, est-il forcément récursivement énumérable ? La réponse est bien évidement non, par un argument de cardinalité similaire à celui vu au chapitre 1 : L'ensemble des programmes informatique capables d'énumérer des ensembles est dénombrable, et il y a un nombre indénombrable d'ensembles. Donc certains d'entre eux ne peuvent pas être récursivement énumérable.

On donne ici une autre preuve, constructive, de ce fait. On définit pour cela l'ensemble des codes de fonctions totales, comme vu dans l'exercice 4.2.5 :

Définition 4.3.1 : On définit l'ensemble des codes de fonctions totales :

$$TOT = \{e \ : \ \forall n \ \Phi_e(n)\downarrow\}$$

Proposition 4.3.2 : On a $\emptyset' \leq_e TOT$. En particulier TOT n'est pas calculable. On a également que TOT n'est pas récursivement énumérable. ⋆

PREUVE: D'après l'exercice 4.2.5 on a $\emptyset' \leq_e TOT$. En particulier TOT n'est pas calculable. Pour montrer que TOT n'est pas récursivement énumérable, il suffit de montrer la chose suivante : $\mathbb{N} - \emptyset' \leq_e TOT$. D'après la proposition 4.2.7 on en déduit que si TOT est récursivement énumérable, alors $\mathbb{N} - \emptyset'$ est lui aussi récursivement énumérable, ce qui n'est pas vrai.

On définit pour ça la fonction totale calculable f telle que :

$$\begin{aligned} \Phi_{f(e)}(n) &= 1 \quad \text{si } \Phi_e(e)[n]\uparrow \\ &= \uparrow \quad \text{sinon} \end{aligned}$$

Il est clair que si $\Phi_e(e)\uparrow$ alors la fonction $\Phi_{f(e)}$ est totale. Par ailleurs si $\Phi_e(e)\downarrow$ alors il existe t tel que $\Phi_e(e)[t]\downarrow$ et la fonction $\Phi_{f(e)}$ ne sera pas totale car $\Phi_{f(e)}(t)\uparrow$. On a donc $\mathbb{N} - \emptyset' \leq_e TOT$ ce qui implique que TOT n'est pas récursivement énumérable. ∎

On quelque sorte TOT est strictement plus puissant que l'arrêt : il peut calculer l'arrêt, mais l'arrêt ne peut pas le calculer. On pose à présent la question inverse : existe-t-il des ensembles non-calculables qui sont strictement moins puissants que l'arrêt : des ensembles qui sont récursivement énumérables, qui ne sont pas calculables, mais tels que l'on a pas $\emptyset' \leq_e A$?

Une version plus forte de cette question (qui fait appel à des notions que nous ne verrons pas durant ce cours) est connu comme le problème de POST. Nous allons montrer que pour ensemble récursivement énumérable et non-calculable B, il existe un ensemble récursivement énumérable et non-calculable A tel que B n'est pas many-one réductible à A.

Notation

On note $\{W_e\}_{e\in\mathbb{N}}$ une énumération des ensembles récursivement énumérable. C'est à dire que $W_e = \{n \ : \ \Phi_e(n)\downarrow\}$.

Notation

On note $W_e[t]$ pour l'ensemble $W_e = \{n \leq t \ : \ \Phi_e(n)[t]\downarrow\}$.

Nous aurons besoin pour montrer notre théorème d'utiliser la notion d'ensemble simple :

Définition 4.3.3 : Un ensemble A est *simple* si son complémentaire est infini et si il intersecte tous les ensembles récursivement énumérables infinis :

$$\forall e \ |W_e| = \infty \rightarrow A \cap W_e \neq \varnothing$$

et

$$|\mathbb{N} - A| = \infty$$

Exercice 4.3.4

Montrez qu'un ensemble simple n'est pas calculable.

Exercice 4.3.5

Montrez que $\emptyset'$ n'est pas un ensemble simple.

Exercice 4.3.6

Montrez qu'il existe un ensemble récursivement énumérable W tel que si $\exists n \in W_e$ avec $n \geq 2e$, alors $W \cap W_e \neq \emptyset$, et tel que pour tout e, l'ensemble W contient au plus e éléments inférieur à $2e$. En déduire qu'il existe un ensemble récursivement énumérable simple.

Nous démontrons à présent le théorème promis.

Théorème 4.3.7:
Soit B un ensemble récursivement énumérable non-calculable. Il existe un ensemble récursivement énumérable non-calculable A tel que $B \not\leq_m A$.

PREUVE: Montrons d'abord la chose suivante : Supposons qu'il existe un ensemble C et une fonction calculable totale f tel que $n \in B$ iff $f(n) \in C$. Alors $|f(\mathbb{N}-B)| = \infty$. Supposons l'inverse, dans le but de montrer que B est calculable. C'est à dire qu'il existe des éléments $x_1, \ldots, x_k \notin C$ tel que $n \notin B \leftrightarrow \exists i \leq k \ f(n) = x_i$. Alors B est calculable par la fonction g suivante :

$$\begin{aligned} g(n) &= 0 \quad \text{si } f(n) = x_0 \\ &\ldots \\ &= 0 \quad \text{si } f(n) = x_k \\ &= 1 \quad \text{sinon} \end{aligned}$$

Comme B est non-calculable, on en déduit que $|f(\mathbb{N} - B)| = \infty$.

Nous allons à présent construire un ensemble récursivement énumérable A tel que $B \not\leq_e A$. On va s'assurer que A est non-calculable en faisant en sorte que A soit un ensemble simple. On va s'assurer que l'on a $B \not\leq_e A$ en faisant en sorte que pour toute fonctions totale f tel que $f(\mathbb{N} - B) = \infty$, on a $n \notin B$ et $f(n) \in A$. Voici plus précisément les conditions, pour tout $e \in \mathbb{N}$:

- R_e^1 : $|W_e| = \infty \rightarrow A \cap W_e \neq \emptyset$
- R_e^2 : Φ_e totale et $|\Phi_e(\mathbb{N} - B)| = \infty \rightarrow \exists n \notin B$ et $\Phi_e(n) \in A$
- R_e^3 : Il y a e entiers $n_1, \ldots n_e$ qui n'appartiennent pas à A.

Les conditions R_e^1 impliquent que A intersecte tous les ensembles récursivement énumérables infinis. Les conditions R_e^3 impliquent que le complémentaire de A est infini. Les conditions R_e^2 impliquent que $B \not\leq_e A$. Notez que conformément à ce que l'on a montré plus haut, on peut se restreindre pour les conditions R_e^2 aux fonctions Φ_e tel que $|\Phi_e(\mathbb{N})| = \infty$ (car B est incalculable). Comment satisfaire chacune de ces conditions ?

Les conditions R_e^1 :
Pour les conditions R_e^1 il suffit d'attendre qu'un élément a soit énuméré dans W_e et de l'énumérer dans A. A ce moment on est sûr que la condition est satisfaite.

Les conditions R_e^2 :
Pour les conditions R_e^2 les choses sont plus difficiles : on ne peut pas choisir un entier

dont on est sûr qu'il n'appartiendra jamais à B pour l'énumérer dans A. Pour cela notre meilleurs possibilité est la suivante : A une étape de calcul t, on prend le premier entier qui n'appartient pas à B pour le moment, et on l'énumère dans A. Si cet entier appartient finalement à B, on finira par le savoir au bout d'un moment, après un temps de calcul t. A ce moment on sait que l'on s'est trompé, et on recommence avec le prochain entier qui n'appartient (pour le moment) pas à B, et ainsi de suite. Comme B est non-calculable, il doit bien exister un élément qui n'appartient pas à B. Aussi on finira-t-on bien par arriver au premier de cet élément dans notre algorithme et à l'énumérer dans A. A ce moment les choses se stabilisent donc pour la condition R_e^2, qui elle est satisfaite parce que quelque chose n'arrive jamais.

Les conditions R_e^3 :
Qu'en est-il des conditions R_e^3 ? A première vu il suffit de choisir e entiers distincts $n_1, \ldots, n_e$ dont on décide qu'ils ne seront jamais énumérés dans A. On voit tout de suite évidement comment cette condition rentre en conflit avec les conditions R_e^1 et R_e^2.

Il sera en effet difficile de satisfaire toutes ces conditions en même temps. On utilise pour cela la méthode dite des priorités, utilisée pour la première fois par Friedberg et Munchnik, pour montrer une version plus forte de ce théorème. L'idée est la suivante : on choisit un ordre sur l'ensemble de toutes les conditions, par exemple l'ordre suivant :

— Les conditions R_{3e} seront les conditions R_e^1
— Les conditions R_{3e+1} seront les conditions R_e^2
— Les conditions R_{3e+2} seront les conditions R_e^3

Ensuite à chaque étape de calcul t, on cherche à satisfaire les t premières conditions dans l'ordre : d'abord la première, puis la deuxième, etc... Que peut-il se passer d'ennuyeux ? Peut-être qu'une condition $R_{e_1}^2$ avait énuméré un entier n dans A à un instant t_1. A un instant $t_2 > t_1$ peut-être qu'une condition $R_{e_2}^3$ avait décidé que $n+1$ n'appartiendrait jamais à A, et peut-être qu'à un instant $t_3 > t_2$ on se rend compte pour la condition $R_{e_1}^2$ que finalement on a $n \in B$. Il se trouve que $n+1$ n'appartient pas à B pour le moment. Pour satisfaire $R_{e_1}^2$ il faut donc énumérer $n+1$ dans A, mais cela rentrerait en conflit avec la condition $R_{e_2}^3$.

On établit donc une règle pour gérer les conflits : Une fois les conditions ordonnées, une condition R_a ne peut jamais déclencher un conflit avec une condition R_b pour $b < a$: plus une condition apparaît tôt dans notre ordre, plus elle est prioritaire sur les autres. En revanche, une condition R_a peut toujours déclencher en conflit avec une condition R_c pour $c > a$: Si, afin d'être satisfaite, la condition R_a doit détruire le plan établi par des conditions moins prioritaires, elle a le droit de le faire.

On décrit à présent l'algorithme qui énumère A : A l'étape de de calcul t, pour tout $a \leq t$:

(a) Soit $A[t-1]$ l'ensemble A énuméré jusqu'ici, c'est à dire jusqu'à l'étape de calcul $t-1$.

(b) Soit k l'entier maximal parmi les entiers qui ont été décidés comme n'appartenant pas à A par des conditions de la forme R_{3e+2} pour $3e+2 < a$.

(c) En fonction de la condition correspondant à a, on fait les choses suivantes :

1. Si $a = 3e$ (correspondant à la condition R_e^1), si $A[t-1] \cap W_e[t] = \varnothing$ et si il existe $n > k$ tel que $n \in W_e[t]$, on énumère n dans A.

2. Si $a = 3e + 1$ (correspondant à la condition R_e^2), on cherche le plus petit $n \leq t$ tel que $n \notin B$ et tel que $\Phi_e(n)[t] > k$. Si on trouve un tel n on l'énumère dans A.
3. Si $a = 3e + 2$ (correspondant à la condition R_e^3), on décide que les e plus petits entiers encore non énumérés dans A ne seront jamais énumérés dans A.

Nous prétendons que l'algorithme décrit ci-dessus énumère un ensemble A tel que pour tout a les conditions R_a sont satisfaites. Comme nous l'avons déjà dit, des conditions peuvent être satisfaite à un moment, puis redevenir non-satisfaite à cause d'une condition plus prioritaire. Toutefois il n'y a toujours qu'un nombre fini de conditions plus prioritaires qu'une autre condition, et c'est ce que nous allons utiliser.

Il est claire qu'il existe t tel que la condition R_0 (correspondant à R_0^1) sera satisfaite pour toujours à partir du temps t : soit $|W_0| < \infty$ auquel cas la condition est satisfaite pour toujours au temps 0. Soit $|W_0| = \infty$ et on énumèrera dans A le premier élément énuméré dans W_0. La condition R_0 sera donc satisfaite à partir du plus petit temps de calcul t tel que quelque chose est énuméré dans W_0.

Supposons que pour $b < a$ il existe un temps t tel que toutes les conditions R_b sont satisfaites pour toujours au temps t. Montrons qu'il existe un temps $s \geq t$ à partir duquel R_a sera satisfaite pour toujours :

Soit $a = 3e$. Comme toutes les conditions prioritaires à a sont satisfaites pour toujours, l'entier maximal k parmi les entiers qui ont été décidés comme n'appartenant pas à A sera le même pour tout $r \geq t$. Supposons $|W_e| = \infty$. Alors il existe $s \geq t$ et $n > k$ tel que $n \in W_e[s]$ et donc $A[s] \cap W_e \neq \varnothing$ et R_a est satisfaite pour toujours.

Soit $a = 3e + 1$. Comme toutes les conditions prioritaires à a sont satisfaites pour toujours, l'entier maximal k parmi les entiers qui ont été décidés comme n'appartenant pas à A sera le même pour tout $r \geq t$. Supposons Φ totale et $|\Phi_e(\mathbb{N} - B)| = \infty$. En particulier il existe un plus petit entier $m \notin B$ tel que $\Phi(m) > k$. Par minimalité de m, il existe un temps de calcul $s \geq t$ tel que tous les entiers $i \leq m$ sont tels que $i \in B[s]$ ou bien $\Phi_e(i) \leq k$. On aura donc $m \in A[s]$. Comme $m \notin B$ on a bien que R_a est satisfaite pour toujours.

Soit $a = 3e + 2$. Comme toutes les conditions prioritaires à a sont satisfaites pour toujours, aucune d'entre elle ne va plus énumérer quelque chose dans A. En particulier aucune de ces conditions n'essaiera d'énumérer un entier parmi les e premiers entiers n'appartenant pas à $A[t]$. Donc R_a sera satisfaite pour toujours.

Par induction on en déduit que toutes les conditions finissent par être satisfaites pour toujours, ce qui conclut la preuve. ∎

Exercice 4.3.8

Deux ensemble A, B avec $A \cap B = \varnothing = 0$ sont récursivement séparables si il existe un ensemble calculable C tel que $A \subseteq C$ et $C \cap B = \varnothing$. Montrez qu'il existe deux ensembles récursivement énumérables A, B qui ne sont pas récursivement séparables.

Exercice 4.3.9

Montrez qu'il existe deux ensembles calculables A, B tel que l'ensemble $C = \{x - y \ : \ x \in A \text{ et } y \in B\}$ est non-calculable.

Chapitre 5

Corrections Exercices

Exercice: 1.2.2

Soit A un ensemble infini. Soit $f : \mathbb{N} \to A$ une surjection. Montrez qu'il existe une bijection $g : \mathbb{N} \to A$.

Preuve: On définit g de la manière suivante :

$$g(n) = f(m) \text{ pour le plus petit } m \text{ tel que } f(m) \notin \{g(i) \ : \ i < n\}$$

Montrons que g est bien défini partout. Soit $n \in \mathbb{N}$. Comme A est infini il doit exister un élément $a \in A$ tel que $a \notin \{g(i) \ : \ i < n\}$. Comme f est une surjection il existe un élément $m \in \mathbb{N}$ tel que $f(m) = a$. Il existe donc bien un élément m (et donc un plus petit tel élément) tel que $f(m) \notin \{g(i) \ : \ i < n\}$. Donc $g(n)$ est bien défini.

Il est clair par définition de g que g est injective. Montrons que g est surjective : Supposons par contradiction qu'il existe un élément $a \in A$ tel que $a \notin g(\mathbb{N})$. Comme f est une surjection, il existe un plus petit élément m tel que $f(m) \notin g(\mathbb{N})$. En particulier, par minimalité de m, pour tout $i < m$ il existe i_n tel que $g(i_n) = f(i)$. Soit $n = \max_{i<m} i_n$.

Comme $f(m) \notin g(\mathbb{N})$ on a donc que $f(m) \notin \{g(j) \ : \ j < n\}$. Par ailleurs, par définition de n, on a pour tout $i \leq m$ que $f(i) \in \{g(j) \ : \ j < n\}$. En particulier m est le plus petit élément tel que $f(m) \notin \{g(j) \ : \ j < n\}$. Par définition de g on a donc $g(n) = f(m)$. Contradiction. ∎

Exercice: 1.2.3

Soit A un ensemble. Soit $f : \mathbb{N} \to A$ une bijection. Soit $B \subseteq A$ un sous-ensemble infini de A. Montrez qu'il existe une bijection $g : \mathbb{N} \to B$.

Preuve: On définit g de la manière suivante :

$$g(n) = f(m) \text{ pour le plus petit } m \text{ tel que } f(m) \in B \text{ et } f(m) \notin \{g(i) \ : \ i < n\}$$

Montrons que g est bien défini partout. Soit $n \in \mathbb{N}$. Comme B est infini, il existe un élément $b \in B$ tel que $b \notin \{g(i) \ : \ i < n\}$. Comme f est une surjection, il existe un élément m tel que $f(m) = b$. Il existe donc bien un élément m (et donc un plus petit tel élément) tel que $f(m) \in B$ et $f(m) \notin \{g(i) \ : \ i < n\}$. Donc $g(n)$ est bien défini.

Il est clair par définition de g que g est injective. Montrons que g est surjective : Supposons par contradiction qu'il existe un élément $b \in B$ tel que $b \notin g(\mathbb{N})$. Comme f est

une surjection, il existe un plus petit élément m tel que $f(m) \in B$ et $f(m) \notin g(\mathbb{N})$. En particulier, par minimalité de m, pour tout $i < m$, soit $f(i) \notin B$, soit il existe i_n tel que $g(i_n) = f(i)$. Soit $n = \max_{i<m} i_n$.

Comme $f(m) \notin g(\mathbb{N})$ on a donc que $f(m) \notin \{g(j) \ : \ j < n\}$. Par ailleurs, par définition de n, on a pour tout $i \leq m$ que $f(i) \notin B$ ou $f(i) \in \{g(j) \ : \ j < n\}$. En particulier m est le plus petit élément tel que $f(m) \in B$ et $f(m) \notin \{g(j) \ : \ j < n\}$. Par définition de g on a donc $g(n) = f(m)$. Contradiction. ∎

Exercice: 1.3.4

Écrivez des programmes goto qui :

1. ne s'arrête pas quelque soit la valeur des registres au début de l'exécution
2. s'arrête ssi dans le registre R_1 se trouve un nombre inférieur ou égale à 2
3. calcule la fonction $a \mapsto 2a$
4. calcule la fonction
$$\begin{array}{rcll} a & \mapsto & 0 & \text{si } a = 0 \\ & \mapsto & 1 & \text{si } a > 0 \end{array}$$
5. calcule la fonction
$$\begin{array}{rcll} (a,b) & \mapsto & 1 & \text{si } a \leq b \\ & \mapsto & 0 & \text{si } a > b \end{array}$$

Preuve:

1. Programme 1 :
   0 goto 0
   8 halt
2. Programme 2 :
   0 $R_1 := R_1 - 1$
   1 $R_1 := R_1 - 1$
   2 if $R_1 = 0$ goto 4
   3 goto 3
   4 halt
3. Programme 3 :
   0 if $R_1 = 0$ goto 5
   1 $R_1 := R_1 - 1$
   2 $R_0 := R_0 + 1$
   3 $R_0 := R_0 + 1$
   4 *goto*0
   5 halt
4. Programme 4 :
   0 if $R_1 = 0$ goto 2
   1 $R_0 := R_0 + 1$
   2 halt
5. Programme 5 :
   0 if $R_1 = 0$ goto 4
   1 if $R_2 = 0$ goto 5
   2 $R_1 := R_1 - 1$
   3 $R_2 := R_2 - 1$
   4 $R_0 := R_0 + 1$
   5 halt

∎

Exercice: 1.3.8

Écrivez un programme structuré qui :

1. calcule la fonction qui a (a, b) associe $a \times b$
2. calcule la fonction qui a a associe le plus grand entier b tel que $b^2 \leq a$
3. calcule la fonction qui a a associe le plus petit entier b tel que $a \leq 2^b$

PREUVE:

Algo1

```
R0 := 0
for i := 0 to R1 do
    for i := 0 to R2 do
        R0 := R0 + 1
    end
end
```

Algo2

```
R2 := 1
R5 := 1
while R5 ≠ 0 do
    R3 := 0
    for i := 0 to R2 do
        for i := 0 to R2 do
            R3 := R3 + 1
        end
    end
    R4 := R1
    R6 := 1
    while R6 ≠ 0 do
        if R3 = 0 then
            R2 := R2 + 1
            R6 := 0
        else
            if R4 = 0 then
                R0 := R2
                R0 := R0 − 1
                R6 := 0
                R5 := 0
            else
                R3 := R3 − 1
                R4 := R4 − 1
            end
        end
    end
end
```

Algo3

```
R2 := 0
R5 := 1
while R5 ≠ 0 do
    R3 := 0
    for i := 0 to R2 do
        for i := 0 to R2 do
            for i := 0 to R2 do
                R3 := R3 + 1
            end
        end
    end
    R4 := R1
    R6 := 1
    while R6 ≠ 0 do
        if R4 = 0 then
            R0 := R2
            R6 := 0
            R5 := 0
        else
            if R3 = 0 then
                R2 := R2 + 1
                R6 := 0
            else
                R3 := R3 − 1
                R4 := R4 − 1
            end
        end
    end
end
```

Exercice: 1.3.9

Ecrivez un programme structuré qui s'arrête ssi le nombre dans le registre R_1 est pair.

Preuve:

Algo

```
R2 := R1
R3 := 1
while R3 ≠ 0 do
    if R2 = 0 then
        R3 = 0
    else
        R2 = R2 − 1
        if R2 = 0 then
            R4 = 1
            while R4 ≠ 0 do
                R4 = 1
            end
        else
            R2 = R2 − 1
        end
    end
end
```

Exercice: 2.1.5

Montrez que les fonctions suivantes sont primitives récursives (réutilisez les fonctions de chaque question pour la question suivante) :

1. La fonction $s_2 : x \mapsto x + 2$
2. La fonction $sp_2 : (x, y) \mapsto y + 2$
3. La fonction $t_2 : x \mapsto 2x$
4. La fonction $f : x \mapsto 2x + 1$

PREUVE:

1. $s_2(x) = \text{succ}(\text{succ}(x))$
2. $sp_2(x, y) = s_2(p_2^2(x, y))$
3. $\begin{array}{lcl} t_2(0) & = & 0 \\ t_2(n+1) & = & sp_2(n, t_2(n)) \end{array}$
4. $f(n) = \text{succ}(t_2(n))$

■

Exercice: 2.1.6

Montrez que l'ensemble des fonctions primitives récursives est clôt par le schéma de définition par itération : pour une fonction $g : \mathbb{N}^a \to \mathbb{N}$, et pour une fonction $h : \mathbb{N}^{a+1} \to \mathbb{N}$, la fonction

$$\begin{array}{rcl} f(x_1, \ldots, x_a, 0) & = & g(x_1, \ldots, x_a, 0) \\ f(x_1, \ldots, x_a, n+1) & = & h(x_1, \ldots, x_a, f(x_1, \ldots, x_a, n)) \end{array}$$

est primitive récursive.

PREUVE: Pour toute fonction primitive récursive $h : \mathbb{N}^{a+1} \to \mathbb{N}$, on peut définir une fonction primitive récursive $h_2 : \mathbb{N}^{a+2} \to \mathbb{N}$ tel que :

$$\forall x_1, \ldots, \forall x_a \ \forall n \ \forall z \ h_2(x_1, \ldots, x_a, n, z) = h(x_1, \ldots, x_a, z)$$

On définie simplement :

$$h_2(x_1, \ldots, x_a, n, z) = h(p_1^{a+2}(x_1, \ldots, x_a, n, z), \ldots, p_a^{a+2}(x_1, \ldots, x_a, n, z), p_{a+2}^{a+2}(x_1, \ldots, x_a, n, z))$$

Donc pour toute fonction $g : \mathbb{N}^a \to \mathbb{N}$, et pour toute fonction $h : \mathbb{N}^{a+1} \to \mathbb{N}$, la fonction

$$\begin{array}{rcl} f(x_1, \ldots, x_a, 0) & = & g(x_1, \ldots, x_a, 0) \\ f(x_1, \ldots, x_a, n+1) & = & h(x_1, \ldots, x_a, f(x_1, \ldots, x_a, n)) \end{array}$$

peut être définie en utilisant le schéma de définition par récurrence, de la manière suivante :

$$\begin{array}{rcl} f(x_1, \ldots, x_a, 0) & = & g(x_1, \ldots, x_a, 0) \\ f(x_1, \ldots, x_a, n+1) & = & h_2(x_1, \ldots, x_a, n, f(x_1, \ldots, x_a, n)) \end{array}$$

Exercice: 2.1.7

Montrez que l'addition, la multiplication et la fonction exponentielle sont primitives récursives.

PREUVE:
Addition :

$$\begin{array}{rcl} \mathrm{add}(a, 0) & = & p_1^2(a, 0) \\ \mathrm{add}(a, b+1) & = & \mathrm{succ}(p_2^2(a, \mathrm{add}(a, b))) \end{array}$$

Multiplication :

$$\begin{array}{rcl} \mathrm{mult}(a, 0) & = & 0 \\ \mathrm{mult}(a, b+1) & = & \mathrm{add}(a, \mathrm{mult}(a, b)) \end{array}$$

Exponentielle :

$$\begin{array}{rcl} \exp(a, 0) & = & 1 \\ \exp(a, b+1) & = & \mathrm{mult}(a, \exp(a, b)) \end{array}$$

Exercice: 2.1.8

Montrez que les fonctions primitives récursives sont closes par schéma de composition étendu. Soit $n, m \in \mathbb{N}^*$. Soit $g : \mathbb{N}^n \to \mathbb{N}$ primitive récursive. Pour tout $i \leq n$ soit $m_i \leq m$, soit $h_i : \mathbb{N}^{m_i} \to \mathbb{N}$ primitive récursive et soit $1 \leq a_1^i < \cdots < a_{m_i}^i \leq m$ des indices. Alors la fonction $f : \mathbb{N}^m \to \mathbb{N}$ définie par :

$$f(x_1, \ldots, x_m) = g(h_1(x_{a_1^1}, \ldots, x_{a_{m_1}^1}), \ldots, h_n(x_{a_1^n}, \ldots, x_{a_{m_n}^n}))$$

est primitive récursive.

PREUVE: Il s'agit simplement de jouer avec le schéma de composition et les projections appropriées. On définit :

$$\begin{aligned} hh_1(x_1,\ldots,x_m) &= h_1(p^m_{a^1_1}(x_1,\ldots,x_m),\ldots,p^m_{a^1_{m_1}}(x_1,\ldots,x_m)) \\ &\ldots \\ hh_n(x_1,\ldots,x_m) &= h_n(p^m_{a^n_1}(x_1,\ldots,x_m),\ldots,p^m_{a^n_{m_n}}(x_1,\ldots,x_m)) \end{aligned}$$

On a donc :

$$f(x_1,\ldots,x_m) = g(hh_1(x_1,\ldots,x_m),\ldots,hh_n(x_1,\ldots,x_m))$$

Exercice: 2.1.9

Montrez que la fonction $sg : \mathbb{N} \to \mathbb{N}$ qui à 0 associe 0 et qui à tous les autres entiers associe 1, est primitive récursive. Montrez que la fonction $\overline{sg} : \mathbb{N} \to \mathbb{N}$ qui à 0 associe 1 et qui à tous les autres entiers associe 0, est primitive récursive.

PREUVE:
sg :
Soit $c^2_1 : \mathbb{N}^2 \to \mathbb{N}$ la fonction constante à deux paramètres qui retourne toujours 1.

$$\begin{aligned} \mathrm{sg}(0) &= 0 \\ \mathrm{sg}(x+1) &= c^2_1(x, \mathrm{sg}(x)) \end{aligned}$$

$\overline{sg}$:
Soit $c^2_0 : \mathbb{N}^2 \to \mathbb{N}$ la fonction constante à deux paramètres qui retourne toujours 0.

$$\begin{aligned} \overline{\mathrm{sg}}(0) &= 1 \\ \overline{\mathrm{sg}}(x+1) &= c^2_0(x, \overline{\mathrm{sg}}(x)) \end{aligned}$$

Exercice: 2.1.10

Montrez que les prédicats de comparaisons $\leq, <, \geq, >, =, \neq$ sont primitifs récursifs.

PREUVE:

$$\begin{aligned} a \leq b &= \mathrm{sg}(\mathrm{succ}(b) - a) \\ a < b &= \mathrm{sg}(b - a) \\ a \geq b &= b \leq a \\ a > b &= b < a \\ a = b &= a \leq b \times b \leq a \\ a \neq b &= \overline{\mathrm{sg}}(a = b) \end{aligned}$$

Exercice: 2.1.11

Montrez que si g est une fonction primitive récursive de $\mathbb{N}^{p+1}$ dans $\mathbb{N}$, alors les fonctions :

$$\begin{aligned} f_1(x_1,\ldots,x_p,n) &= \textstyle\sum_{i=0}^{n} g(x_1,\ldots,x_p,i) \\ &\text{et} \\ f_2(x_1,\ldots,x_p,n) &= \textstyle\prod_{i=0}^{n} g(x_1,\ldots,x_p,i) \end{aligned}$$

sont primitives récursives.

PREUVE: On définit la fonction primitive récursive gplus comme :

$$\text{gplus}(x_1,\ldots,x_p,n,z) = g(x_1,\ldots,x_p,\text{succ}(n)) + z$$

On a alors :

$$\begin{aligned} f_1(x_1,\ldots,x_p,0) &= g(x_1,\ldots,x_p,0) \\ f_1(x_1,\ldots,x_p,n+1) &= \text{gplus}(x_1,\ldots,x_p,n,f_1(x_1,\ldots,x_p,n)) \end{aligned}$$

On définit la fonction primitive récursive gmult comme :

$$\text{gmult}(x_1,\ldots,x_p,n,z) = g(x_1,\ldots,x_p,\text{succ}(n)) \times z$$

On a alors :

$$\begin{aligned} f_2(x_1,\ldots,x_p,0) &= g(x_1,\ldots,x_p,0) \\ f_2(x_1,\ldots,x_p,n+1) &= \text{gmult}(x_1,\ldots,x_p,n,f_2(x_1,\ldots,x_p,n)) \end{aligned}$$

Exercice: 2.1.12

Montrez que les fonctions :

1. $\text{pred} : \mathbb{N} \to \mathbb{N}$ qui vaut 0 en 0 et $n-1$ en $n > 0$.
2. $- : \mathbb{N}^2 \to \mathbb{N}$ qui vaut $\max(0, a-b)$

sont primitives récursives.

PREUVE:
La fonction pred :

$$\begin{aligned} \text{pred}(0) &= 0 \\ \text{pred}(n+1) &= p_1^2(n,\text{pred}(n)) \end{aligned}$$

La fonction de soustraction :

$$\begin{aligned} -(a,0) &= p_1^2(a,0) \\ -(a,b+1) &= \text{pred}(p_2^2(a,-(a,b))) \end{aligned}$$

Exercice: 2.1.13

Soit $P_1(a_1,\ldots,a_n)$ et $P_2(b_1,\ldots,b_m)$ des prédicats primitifs récursifs.

1. Montrez que le prédicat $P(a_1,\ldots,a_n,b_1,\ldots,b_m)$ qui est vrai ssi $P_1(a_1,\ldots,a_n)$ est vrai et $P_2(b_1,\ldots,b_m)$ est vrai , est primitif récursif.
2. Montrez que le prédicat $P(a_1,\ldots,a_n,b_1,\ldots,b_m)$ qui est vrai ssi $P_1(a_1,\ldots,a_n)$ est vrai ou $P_2(b_1,\ldots,b_m)$ est vrai , est primitif récursif.
3. Montrez que le prédicat $P(a_1,\ldots,a_n)$ qui est vrai ssi $P_1(a_1,\ldots,a_n)$ est faux, est primitif récursif.

PREUVE:

$$\begin{aligned} P_1(a_1,\ldots,a_n) \text{ and } P_2(b_1,\ldots,b_m) &= P_1(a_1,\ldots,a_n) \times P_2(b_1,\ldots,b_m) \\ P_1(a_1,\ldots,a_n) \text{ or } P_2(b_1,\ldots,b_m) &= \text{sg}(P_1(a_1,\ldots,a_n) + P_2(b_1,\ldots,b_m)) \\ \neg P_1(a_1,\ldots,a_n) &= \overline{\text{sg}}(P_1(a_1,\ldots,a_n)) \end{aligned}$$

Exercice: 2.1.14

Montrez que les prédicats primitifs récursifs sont clôt par quantification existentielle et universelle bornée : Si $P(x_1, \ldots, x_n, z)$ est un prédicat primitif récursif, alors les prédicats :

$$\begin{aligned} Q_1(x_1, \ldots, x_n, z) &= \exists t \leq z \; P(x_1, \ldots, x_n, z) \\ Q_2(x_1, \ldots, x_n, z) &= \forall t \leq z \; P(x_1, \ldots, x_n, z) \end{aligned}$$

sont primitifs récursifs.

PREUVE:

$$\begin{aligned} Q_1(x_1, \ldots, x_n, 0) &= P(x_1, \ldots, x_n, 0) \\ Q_1(x_1, \ldots, x_n, z+1) &= P(x_1, \ldots, x_n, z+1) \text{ or } Q_1(x_1, \ldots, x_n, z) \\ Q_2(x_1, \ldots, x_n, 0) &= P(x_1, \ldots, x_n, 0) \\ Q_2(x_1, \ldots, x_n, z+1) &= P(x_1, \ldots, x_n, z+1) \text{ and } Q_2(x_1, \ldots, x_n, z) \end{aligned}$$

Exercice: 2.1.15

Montrez que l'ensemble des fonctions primitives récursives est clôt par définition par cas sur un prédicat primitif récursif : si g et h sont deux fonction primitives récursives de $\mathbb{N}^p$ dans $\mathbb{N}$, et si P est un prédicat primitif récursif sur $\mathbb{N}^p$, alors la fonction :

$$\begin{aligned} f(x_1, \ldots, x_p) &= g(x_1, \ldots, x_p) && \text{si } P(x_1, \ldots, x_p) \\ f(x_1, \ldots, x_p) &= h(x_1, \ldots, x_p) && \text{sinon} \end{aligned}$$

est primitive récursive.

PREUVE: Comme P est un prédicat primitif récursif, il existe une fonction $d : \mathbb{N}^n \to \mathbb{N}$ tel que

$$\begin{aligned} d(x_1, \ldots, x_p) &= 1 && \text{si } P(x_1, \ldots, x_p) \\ d(x_1, \ldots, x_p) &= 0 && \text{sinon} \end{aligned}$$

On définit donc :

$$f(x_1, \ldots, x_p) = g(x_1, \ldots, x_p) \times \mathrm{sg}(d(x_1, \ldots, x_p)) + h(x_1, \ldots, x_p) \times \overline{\mathrm{sg}}(d(x_1, \ldots, x_p))$$

Exercice: 2.1.16

Montrez que les fonctions primitives récursives sont closes par minimisation bornée : si $f : \mathbb{N}^{p+1} \to \mathbb{N}$ est primitive récursive, alors la fonction

$$\begin{aligned} g(x_1, \ldots, x_p, n) &= \min\{t \leq n \mid f(x_1, \ldots, x_p, t) = 0\} + 1 && \text{si } \exists t \leq n \; f(x_1, \ldots, x_p, t) = 0 \\ &= 0 && \text{sinon} \end{aligned}$$

est primitive récursive.

PREUVE:

$$\begin{aligned} g(x_1, \ldots, x_p, 0) &= 1 && \text{si } f(x_1, \ldots, x_p, 0) = 0 \\ &= 0 && \text{sinon} \\ g(x_1, \ldots, x_p, n+1) &= \mathrm{succ}(\mathrm{succ}(n)) && \text{si } f(x_1, \ldots, x_p, \mathrm{succ}(n)) = 0 \\ &= g(x_1, \ldots, x_p, n) && \text{sinon} \end{aligned}$$

Exercice: 2.2.3

Montrez que pour tout prédicat $P \subseteq \mathbb{N}^{k+1}$, la fonction suivante est récursive :

$$\begin{array}{rcll} f(x_1,\ldots,x_k) & = & \min\{z \ : \ P(z,x_1,\ldots,x_k)\} & \text{si } \exists z \ P(z,x_1,\ldots,x_k) \\ & = & \text{non défini} & \text{sinon} \end{array}$$

Preuve: On rappelle que l'on a une fonction g tel que $g(z,x_1,\ldots,x_k) = 1$ ssi $P(z,x_1,\ldots,x_k)$ et $g(z,x_1,\ldots,x_k) = 0$ ssi $\neg P(z,x_1,\ldots,x_k)$. On a donc :

$$f(x_1,\ldots,x_k) = \mu z.\overline{\text{sg}}(g(z,x_1,\ldots,x_k)) = 0$$

et f st donc bien récursive. ∎

Exercice: 2.2.4

Montrez que pour tout prédicat $P \subseteq \mathbb{N}^{k+1}$ et pour toute fonction récursive $g : \mathbb{N}^k \to \mathbb{N}$, la fonction suivante est récursive :

$$\begin{array}{rcll} f(x_1,\ldots,x_k) & = & g(x_1,\ldots,x_k) & \text{si } \exists z \ P(z,x_1,\ldots,x_k) \\ & = & \text{non défini} & \text{sinon} \end{array}$$

Preuve: On rappelle que l'on a une fonction g tel que $g(z,x_1,\ldots,x_k) = 1$ ssi $P(z,x_1,\ldots,x_k)$ et $g(z,x_1,\ldots,x_k) = 0$ ssi $\neg P(z,x_1,\ldots,x_k)$. On a donc :

$$f(x_1,\ldots,x_k) = g(x_1,\ldots,x_k) \times \text{sg}(\mu z.P(z,x_1,\ldots,x_k) + 1)$$

et f est donc bien récursive. Notez que la multiplication sert à propager la partialité si jamais aucun élément z tel que $P(z,x_1,\ldots,x_k)$ n'existe. ∎

Exercice: 2.2.5

Soit $a_1,\ldots,a_k \in \mathbb{N}$. Montrez que la fonction nulle en $a_1,\ldots,a_k$ et non définie ailleurs, est récursive.

Preuve:

$$\begin{array}{rcll} f(x) & = & 0 & \text{si } x = a_1 \vee \cdots \vee x = a_k \\ & = & \mu z.\text{succ}(z) = 0 & \text{sinon} \end{array}$$

Exercice: 2.2.6

Montrez que la fonction pre, non définit en 0 et vérifiant $\text{pre}(x+1) = x$ est récursive.

Preuve:

$$\begin{array}{rcll} \text{pre}(x) & = & x - 1 & \text{si } x > 0 \\ & = & \mu z.\text{succ}(z) = 0 & \text{sinon} \end{array}$$

Exercice: 2.3.3

Montrez que les fonctions constantes, tout comme les fonctions de projections et la fonction successeur sont calculables par des programmes structurés (et donc par des programmes `goto`).

PREUVE:

c_a^n
$R_0 := a$

p_i^n
$R_0 := R_i$

succ
$R_0 := R_1$ $R_0 := R_0 + 1$

Exercice: 2.4.1

Soit $\alpha_2 : \mathbb{N}^2 \to \mathbb{N}$ la bijection de Cantor définie par :

$$\begin{aligned} \alpha_2(x,y) &= y + \textstyle\sum_{i=0}^{x+y} i \\ &= y + \frac{(x+y+1)(x+y)}{2} \end{aligned}$$

1. Montrez que α_2 est bijective.
2. Montrez que α_2 est primitive récursive.
3. Montrez que $\alpha_2(a,b) \geq a$ et $\alpha_2(a,b) \geq b$.
4. Montrez que les deux projections π_1, π_2 tel que :
   - $\alpha_2(\pi_1(c), \pi_2(c)) = c$
   - $\pi_1(\alpha_2(a,b)) = a$
   - $\pi_2(\alpha_2(a,b)) = b$ sont primitives récursives.

PREUVE:

1. Montrons que $(x_1, y_1) \neq (x_2, y_2)$ implique $\alpha_2(x_1, y_1) \neq \alpha_2(x_2, y_2)$. Supposons $(x_1, y_1) \neq (x_2, y_2)$. Supposons d'abord que $x_1 + y_1 < x_2 + y_2$ (le cas $x_2 + y_2 < x_1 + y_1$ étant symétrique). Alors

$$\begin{array}{rrcl} & (\sum_{i=0}^{x_1+y_1} i) + x_1 + y_1 + 1 & \leq & \sum_{i=0}^{x_2+y_2} i \\ \to & y_1 + \sum_{i=0}^{x_1+y_1} i & < & \sum_{i=0}^{x_2+y_2} i \\ \to & y_1 + \sum_{i=0}^{x_1+y_1} i & < & y_2 + \sum_{i=0}^{x_2+y_2} i \\ \to & \alpha_2(x_1, y_1) & < & \alpha_2(x_2, y_2) \end{array}$$

Supposons à présent que $x_1 + y_1 = x_2 + y_2$. Notez que comme $(x_1, y_1) \neq (x_2, y_2)$, si $y_1 = y_2$, alors nécéssairement $x_1 = x_2$. On a donc forcément que $y_1 \neq y_2$. On a donc :

$$\begin{array}{rrcl} & (\sum_{i=0}^{x_1+y_1} i) & = & \sum_{i=0}^{x_2+y_2} i \\ \rightarrow & y_1 + (\sum_{i=0}^{x_1+y_1} i) & \neq & y_2 + \sum_{i=0}^{x_2+y_2} i \\ \rightarrow & \alpha_2(x_1, y_1) & \neq & \alpha_2(x_2, y_2) \end{array}$$

Montrons à présent que α_2 est surjective. Pour cela montrons d'abord que :

$$\begin{array}{rcll} \alpha_2(x-1, y+1) - \alpha_2(x, y) & = & 1 & \text{pour } x > 0 \\ \alpha_2(y+1, 0) - \alpha_2(0, y) & = & 1 & \end{array}$$

Dans le premier cas on a :

$$\begin{array}{rcl} \alpha_2(x-1, y+1) - \alpha_2(x, y) & = & (y + 1 + \sum_{i=0}^{x-1+y+1} i) - (y + \sum_{i=0}^{x+y} i) \\ & = & (y + 1 + \sum_{i=0}^{x+y} i) - (y + \sum_{i=0}^{x+y} i) \\ & = & 1 \end{array}$$

Dans le deuxième cas on a :

$$\begin{array}{rcl} \alpha_2(y+1, 0) - \alpha_2(0, y) & = & (\sum_{i=0}^{y+1} i) - (y + \sum_{i=0}^{y} i) \\ & = & (y + 1 + \sum_{i=0}^{y} i) - (y + \sum_{i=0}^{y} i) \\ & = & 1 \end{array}$$

Montrons à présent par récurrence que pour tout n, il existe (x, y) tel que $\alpha_2(x, y) = n$. On a bien $\alpha_2(0, 0) = 0$. Supposons à présent que $\alpha_2(x, y) = n$. Si jamais $x > 0$ alors on a donc $\alpha_2(x-1, y+1) = n+1$. Si jamais $x = 0$ on a donc $\alpha_2(y+1, 0) = n+1$. On en déduit que α_2 est surjective.

2. La fonction

$$\begin{array}{rcll} g(x) & = & \{\min z \leq x \text{ tel que } z * 2 = x\} + 1 & \text{si } \exists z \leq x \text{ tel que } z * 2 = x \\ & = & 0 & \text{sinon} \end{array}$$

est primitive récursive (schéma de minimisation bornée). La fonction $\alpha_2(x, y)$ est donnée par :

$$\alpha_2(x, y) = y + g((x + y + 1)(x + y)) - 1$$

3. Trivial
4. En utilisant que $\alpha(a, b) \geq a$ et $\alpha(a, b) \geq b$, on définit $\pi_1(c)$ et $\pi_2(c)$ par schéma de minimisation borné, couplé avec une quantification existentielle bornée :

$$\begin{array}{rcl} \pi_1(c) & = & \min\{a \leq c \ : \ \exists b \leq c \ \alpha_2(a, b) = c\} \\ \pi_2(c) & = & \min\{b \leq c \ : \ \exists a \leq c \ \alpha_2(a, b) = c\} \end{array}$$

Exercice: 2.4.2

On définie par récurrence sur $k \geq 2$ les fonctions :

$$\alpha_{k+1}(x_1, x_2, \ldots, x_{k+1}) = \alpha_2(x_1, \alpha_k(x_2, \ldots, x_{k+1}))$$

1. Montrez que pour tout $k \geq 2$ la fonction α_k est une bijection de $\mathbb{N}^k$ dans $\mathbb{N}$.
2. Montrez que pour tout $k \geq 2$ la fonction α_k est primitive récursive.
3. Montrez que pour tout $k \geq 2$ les projections $\pi_1^k, \ldots, \pi_k^k$ tels que

$$\alpha_k(\pi_1^k(c), \ldots, \pi_k^k(c)) = c$$

sont primitives récursives.

PREUVE:

1. C'est déjà prouvé pour $k = 2$ dans l'exercice précédent. Supposons que α_k soit une injection et montrons que α_{k+1} est une injection. Supposons $(x_1, \ldots, x_{k+1}) \neq (y_1, \ldots, y_{k+1})$. Montrons que l'on a forcément

$$(x_1, \alpha_k(x_2, \ldots, x_{k+1})) \neq (y_1, \alpha_k(y_2, \ldots, y_{k+1}))$$

   Supposons d'abord que $x_1 \neq y_1$. Alors on a clairement $(x_1, \alpha_k(x_2, \ldots, x_{k+1})) \neq (y_1, \alpha_k(y_2, \ldots, y_{k+1}))$. Supposons à présent $(x_2, \ldots, x_{k+1}) \neq (y_2, \ldots, y_{k+1})$. Par hypothèse de récurrence on a que α_k est injective, et donc que $\alpha_k(x_2, \ldots, x_{k+1}) \neq \alpha_k(y_2, \ldots, y_{k+1})$. On a donc $(x_1, \alpha_k(x_2, \ldots, x_{k+1})) \neq (y_1, \alpha_k(y_2, \ldots, y_{k+1}))$.
   A présent, comme α_2 est injective, on a donc que $\alpha_2(x_1, \alpha_k(x_2, \ldots, x_{k+1})) \neq \alpha_2(y_1, \alpha_k(y_2, \ldots, y_{k+1}))$ et donc que $\alpha_{k+1}(x_1, x_2, \ldots, x_{k+1}) \neq \alpha_{k+1}(x_1, x_2, \ldots, x_{k+1})$. On en déduit que α_{k+1} est injective. On en déduit que pour tout $k \geq 2$ la fonction α_k est une injection.
   Supposons à présent que α_k soit une surjection et montrons que α_{k+1} est une surjection. Considérons n. Comme α_2 est une surjection, il existe x_1, y tel que $\alpha_2(x_1, y) = n$. A présent comme α_k est une surjection, il existe $(x_2, \ldots, x_{k+1})$ tel que $\alpha_2(x_2, \ldots, x_{k+1}) = y$. On a donc $\alpha_k(x_1, x_2, \ldots, x_{k+1}) = n$. Comme c'est vrai pour tout n on a donc que α_{k+1} est une surjection. Par récurrence, pour tout $k \geq 2$ on a que α_k est une surjection.
2. α_2 est primitive récursive. Supposons α_k primitive récursive. Alors

$$\alpha_{k+1}(x_1, x_2, \ldots, x_{k+1}) = \alpha_2(x_1, \alpha_k(x_2, \ldots, x_{k+1}))$$

   est clairement primitive récursive. On a donc par récurrence que que pour tout $k \geq 2$ la fonction α_k est primitive récursive.
3. Les projection π_1^2, π_2^2 (aussi notées π_1, π_2) sont primitives récursives. Supposons que les projections $\pi_1^k, \ldots, \pi_k^k$ sont primitives récursives. Montrons que les projections $\pi_1^{k+1}, \ldots, \pi_{k+1}^{k+1}$ sont primitives récursives. On a

$$\begin{array}{rcll} \pi_1^{k+1}(c) & = & \pi_1^2(c) & \\ \pi_{m+1}^{k+1}(c) & = & \pi_m^k(\pi_2^2(c)) & \text{pour } m \leq k \end{array}$$

   Ces projections sont primitives récursives, par hypothèse de récurrence. Donc pour tous k, on a que les projections $\pi_1^{k+1}, \ldots, \pi_{k+1}^{k+1}$ sont primitives récursives.

Exercice: 2.4.3

Utilisez les fonctions α_k pour montrer que si les fonctions $g_1, \ldots, g_k$ de $\mathbb{N}^n$ dans $\mathbb{N}$ et les fonctions $h_1, \ldots, h_n$ de $\mathbb{N}^{n+k+1}$ dans $\mathbb{N}$ sont primitives récursives, alors les fonctions :

$$\begin{array}{rcl} f_1(x_1, \ldots, x_n, 0) & = & g_1(x_1, \ldots, x_n) \\ f_1(x_1, \ldots, x_n, x+1) & = & h_1(x_1, \ldots, x_n, x, f_1(x_1, \ldots, x_n, x), \ldots, f_k(x_1, \ldots, x_n, x)) \end{array}$$

$\ldots$

$$\begin{array}{rcl} f_k(x_1, \ldots, x_n, 0) & = & g_k(x_1, \ldots, x_n) \\ f_k(x_1, \ldots, x_n, x+1) & = & h_k(x_1, \ldots, x_n, x, f_1(x_1, \ldots, x_n, x), \ldots, f_k(x_1, \ldots, x_n, x)) \end{array}$$

sont primitives récursives.

PREUVE: On définit (avec $\overline{x} = x_1, \ldots, x_k$) :

$$\begin{array}{rcll} f(\overline{x}, 0) & = & \langle & g_1(\overline{x}), \ldots, g_k(\overline{x})\rangle \\ f(\overline{x}, x+1) & = & \langle & h_1(\overline{x}, x, \pi_1(f(\overline{x}, x)), \ldots, \pi_k(f(\overline{x}, x))), \\ & & & \ldots, \\ & & & h_k(\overline{x}, x, \pi_1(f(\overline{x}, x)), \ldots, \pi_k(f(\overline{x}, x)))\rangle \end{array}$$

On a que $f_k(x_1, \ldots, x_n, x) = \pi_k(f(x_1, \ldots, x_n, x))$ ∎

Exercice: 2.4.4

Soit $::$ une notation pour la fonction primitive récursive $a :: l = 1 + \alpha_2(a, l)$. Soit hd et tl les fonctions définies par :

$$\begin{array}{rclrcl} \mathrm{hd}(0) & = & 0 & \mathrm{tl}(0) & = & 0 \\ \mathrm{hd}(x :: l) & = & x & \mathrm{tl}(x :: l) & = & l \end{array}$$

On définie la notation [] pour représenter le codage suivant des listes d'entiers :

$$\begin{array}{rcl} [\,] & = & 0 \\ [a_0, a_1, \ldots, a_n] & = & a_0 :: [a_1, \ldots, a_n] \end{array}$$

1. Montrez que la fonction [] est bijective.
2. Montrez que les fonctions hd et tl sont primitives récursives.
3. Montrez que la fonction gett : $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$ telle que $\mathrm{gett}([x_0, \ldots, x_n], i) = [x_i, \ldots, x_n]$ est primitive récursive.
4. Montrez que la fonction get : $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$ telle que $\mathrm{get}([x_0, \ldots, x_n], i) = x_i$ est primitive récursive.

PREUVE: 1. Montrons par récurrence que deux listes de taille différentes ne peuvent pas être codées par le même élément. Montrons d'abord que le code de la liste vide est toujours différent du code d'une liste non vide. Le code de la liste vide est 0. le code d'une liste non vide est de la forme $1 + \alpha_2(a, l)$ et est donc différent de 0, ce qui prouve que le code de la liste vide est toujours différent du code d'une liste non vide. Supposons à présent que toute les listes de taille n ont un code différent de celui des listes de taille $m > n$. Montrons que toutes les listes de taille $n + 1$ ont un coe différent de celui des listes de taille $m > n + 1$.

Les codes des listes de taille $n+1$ sont de la forme $1 + \alpha_2(a, l_1)$ pour l_1 le code d'une liste de taille n. Le codes des listes de taille $m > n+1$ sont de la forme $1+\alpha_2(a, l_2)$ pour l_2 le code d'une liste de taille $m > n$. Par hypothèse de récurrence, on a forcément $l_1 \neq l_2$ et comme α_2 est injectif, on a forcément $1 + \alpha_2(a, l_1) \neq 1 + \alpha_2(a, l_2)$. Donc les codes des listes de taille $n + 1$ sont différents des codes des listes de taille $m > n + 1$. Par récurrence, on en déduit que les codes de listes de taille différente sont différents.

Montrons à présent par réccurence sur k que si $(a_1, \ldots, a_k) \neq (b_1, \ldots, b_k)$, alors on a $[a_1, \ldots, a_k] \neq [b_1, \ldots, b_k]$. Pour $k = 1$ on a que $a_1 \neq b_1$ implique $1 + \alpha_2(a_1, 0) \neq 1 + \alpha_2(b_1, 0)$ car α_2 est injective. On a donc $[a_1] \neq [b_1]$. Supposons que ce soit le cas pour k et montrons que c'est le cas pour $k + 1$. Supposons $(a_1, \ldots, a_{k+1}) \neq (b_1, \ldots, b_{k+1})$. Si $a_1 \neq b_1$, alors on a $1 + \alpha_2(a_1, [a_2, \ldots, a_{k+1}]) \neq 1 + \alpha_2(b_1, [b_2, \ldots, b_{k+1}])$ car α_2 est injective. Si $(a_2, \ldots, a_{k+1}) \neq (b_2, \ldots, b_{k+1})$,

alors par hypothès de récurrence on a $[a_2,\ldots,a_{k+1}] \neq [b_2,\ldots,b_{k+1}]$ et donc $1+\alpha_2(a_1,[a_2,\ldots,a_{k+1}]) \neq 1+\alpha_2(b_1,[b_2,\ldots,b_{k+1}])$ car α_2 est injective. Dans tous les cas on a $[a_1,\ldots,a_{k+1}] \neq [b_1,\ldots,b_{k+1}]$. Par récurrence, pour tout k on a donc $(a_1,\ldots,a_k) \neq (b_1,\ldots,b_k)$ implique $[a_1,\ldots,a_k] \neq [b_1,\ldots,b_k]$. On a donc que la fonction [] est injective.

Montrons à présent que [] est surjective. Supposons par contradiction que ce n'est pas le cas. Dans ce cas il existe un plus petit n tel que n n'est le code d'aucune liste. Notez que l'on a forcément $n > 0$ car 0 est le code de la liste vide. Aussi comme α_2 est surjective, il existe (a,b) tel que $\alpha_2(a,b) = n-1$ et donc tel que $1+\alpha_2(a,b) = n$. On a par ailleurs nécéssairement que $b \leq n-1 < n$. Aussi par minimalité de n, il doit exister une liste l dont b est le code. On a alors que n est le code de la liste $a :: l$, ce qui contredit que n n'est le code d'aucune liste. La fonction [] est donc surjective.

2. On a :
$$\begin{array}{rclcrcl} \mathrm{hd}(0) & = & 0 & & \mathrm{tl}(0) & = & 0 \\ \mathrm{hd}(l) & = & \pi_1(l-1) & & \mathrm{tl}(l) & = & \pi_2(l-1) \end{array}$$

3. On a :
$$\begin{array}{rcl} \mathrm{gett}(l,0) & = & l \\ \mathrm{gett}(l,n+1) & = & \mathrm{tl}(\mathrm{gett}(l,n)) \end{array}$$

4. On a :
$$\mathrm{get}(l,n) = \mathrm{hd}(\mathrm{gett}(l,n))$$

Exercice: 3.2.1

1. Montrez que les fonctions primitives récursives sont closes par récurrence primitive sur la suite des valeurs : si $g : \mathbb{N}^a \to \mathbb{N}$ et $h : \mathbb{N}^{a+2} \to \mathbb{N}$ sont primitives récursives, alors la fonction $f : \mathbb{N}^{a+1} \to \mathbb{N}$ définit par :
$$\begin{array}{rcl} f(x_1,\ldots,x_a,0) & = & g(x_1,\ldots,x_a) \\ f(x_1,\ldots,x_a,n+1) & = & h(x_1,\ldots,x_a,n,[f(x_1,\ldots,x_a,n),\ldots,f(x_1,\ldots,x_a,0)]) \end{array}$$
est primitive récursive.

2. Montrez que si $g : \mathbb{N}^a \to \mathbb{N}$ et $h : \mathbb{N}^{a+2} \to \mathbb{N}$ sont primitives récursives, et si $d_1,\ldots,d_k : \mathbb{N} \to \mathbb{N}$ sont primitives récursives et telles que $d_i(n) \leq n$ pour tout n et $i \leq k$, alors la fonction $f : \mathbb{N}^{a+1} \to \mathbb{N}$ définit par :
$$\begin{array}{rcll} f(x_1,\ldots,x_a,0) & = & g(x_1,\ldots,x_a) & \\ f(x_1,\ldots,x_a,n+1) & = & h(x_1,\ldots,x_a,n, & f(x_1,\ldots,x_a,d_1(n)), \\ & & & \ldots, \\ & & & f(x_1,\ldots,x_a,d_k(n))) \end{array}$$
est primitive récursive.

Preuve:

1. On définit $t : \mathbb{N}^{a+1} \to \mathbb{N}$ de la manière suivante :
$$\begin{array}{rcl} t(x_1,\ldots,x_a,0) & = & g(x_1,\ldots,x_a) :: [] \\ t(x_1,\ldots,x_a,n+1) & = & h(x_1,\ldots,x_a,n,t(x_1,\ldots,x_a,n)) :: t(x_1,\ldots,x_a,n) \end{array}$$
Il est clair que $\mathrm{hd}(t(x_1,\ldots,x_a,n))$ est égale à $f(x_1,\ldots,x_a,n)$ définit par :
$$\begin{array}{rcl} f(x_1,\ldots,x_a,0) & = & g(x_1,\ldots,x_a) \\ f(x_1,\ldots,x_a,n+1) & = & h(x_1,\ldots,x_a,n,[f(x_1,\ldots,x_a,n),\ldots,f(x_1,\ldots,x_a,0)]) \end{array}$$

On a donc en particulier que f est primitive récursive.

2. On définit $f : \mathbb{N}^{a+1} \to \mathbb{N}$ de la manière suivante (avec $\overline{x} = x_1, \ldots, x_a$) :

$$\begin{array}{rcll} f(\overline{x}, 0) & = & g(\overline{x}) & \\ f(\overline{x}, n+1) & = & h(\overline{x}, n, & \text{get}([f(\overline{x}, n), \ldots, f(\overline{x}, 0)], n - d_0(n)), \\ & & & \ldots, \\ & & & \text{get}([f(\overline{x}, n), \ldots, f(\overline{x}, 0)], n - d_k(n))) \end{array}$$

Exercice: 3.2.2

On défini la notation @ pour une fonction de $\mathbb{N}^2$ vers $\mathbb{N}$, telle que

$$[x_0, \ldots, x_n]@[y_0, \ldots, y_m] = [x_0, \ldots, x_n, y_0, \ldots, y_m]$$

Montrez que la fonction $@ : \mathbb{N}^2 \to \mathbb{N}$ est primitive récursive.

PREUVE: La fonction suivante $@([y_0, \ldots, y_m], [x_0, \ldots, x_n])$ sera égale à $[x_0, \ldots, x_n, y_0, \ldots, y_m]$:

$$\begin{array}{rcl} @([y_0, \ldots, y_m], 0) & = & [y_0, \ldots, y_m] \\ @([y_0, \ldots, y_m], n+1) & = & \text{hd}(\text{succ}(n)) :: @([y_0, \ldots, y_m], \text{tl}(\text{succ}(n))) \end{array}$$

On a bien que $\text{tl}(\text{succ}(n)) \leq n$ pour tout n et donc que @ est bien définit avec le deuxième schéma de l'exercice 3.2.1.

Exercice: 3.3.2

Soit $A \subseteq \mathbb{N}$ tel que $A \neq \mathbb{N}$ et A est calculable.

1. Montrez qu'il existe une fonction récursive f telle que $x \in A$ iff $f(x) \notin A$.
2. En utilisant le théorème du point fixe, en déduire qu'il existe i, j tel que $\Phi_i = \Phi_j$ avec $i \in A$ et $j \notin A$.
3. En déduire qu'il n'existe pas de prédicats calculables P tel que $P(e)$ ssi e est le code d'un programme qui fait la multiplication par 2.
4. Généralisez la question précédente pour montrer le théorème de Rice : Pour tout comportement non-trivial des programmes, il n'est pas possible de calculer l'ensemble des codes de programmes ayant ce comportement. Formellement : soit un ensemble $C \neq \mathbb{N}$ de codes de fonctions tel que pour tout e_1, e_2 avec $\phi_{e_1} = \phi_{e_2}$, on a $e_1 \in C \to e_2 \in C$. Alors C n'est pas calculable.

Exercice: 3.4.2

Montrez que pour tout n, la fonction A_n est primitive récursive.

PREUVE: Rappelons la définition de A_n :

$$\begin{array}{rcl} A_0(x) & = & 2^x \\ A_{n+1}(0) & = & 1 \\ A_{n+1}(x) & = & A_n(A_{n+1}(x-1)) \end{array}$$

Il est clair que A_0 est primitive récursive. Supposons que A_n soit primitive récursive, et montrons que A_{n+1} est primitive récursive. On a :

$$\begin{array}{rcl} A_{n+1}(0) & = & 1 \\ A_{n+1}(x) & = & A_n(A_{n+1}(x-1)) \end{array}$$

ce qui est bien une définition primitive récursive. Par récurrence on a donc que pour tout n, la fonction A_n est primitive récursive. ∎

Exercice: 4.1.3

Montrez que tout ensemble calculable est récursivement énumérable.

PREUVE: Supposons qu'il existe $e \in \mathbb{N}$ tel que $n \in A \leftrightarrow \Phi_e(n) \downarrow = 1$ et $n \notin A \leftrightarrow \Phi_e(n) \downarrow = 0$. On construit la fonction récursive suivante :

$$\begin{array}{rcll} \Phi_a(n) & = & 1 & \text{si } \Phi_e(n) \downarrow = 1 \\ & = & \uparrow & \text{sinon} \end{array}$$

On a que $n \in A$ ssi $\Phi_a(n) \downarrow$. Donc A est récursivement énumérable. ∎

Exercice: 4.1.8

Un ensemble infini est récursivement énumérable dans l'ordre si il existe une fonction totale $f : \mathbb{N} \to \mathbb{N}$ telle que $f(\mathbb{N}) = A$ et telle que $f(n) < f(n+1)$. Montrez que si un ensemble infini A est récursivement énumérable dans l'ordre, alors A est calculable.

PREUVE: On définit :

$$\begin{array}{rcll} b(n) & = & \mu z.f(z) \geq n & \\ g(n) & = & 1 & \text{si } f(b(n)) = n \\ & = & 0 & \text{sinon} \end{array}$$

Notons que b est une fonction totale, car $f(\mathbb{N}) = A$ et A est infini. Donc pour tout n il existe z tel que $f(z) \geq n$. Supposons à présent que $n \in A$. Dans ce cas $b(n)$ sera égale à z tel que $f(z) = n$. En particulier $f(b(n)) = n$ et donc $g(n) = 1$. A présent si $n \notin A$, alors $b(n)$ sera égale à z tl que $f(z) > n$. En particulier on aura $f(b(n)) > n$ et donc $g(n) = 0$. ∎

Exercice: 4.1.9

Montrez que tout ensemble récursivement énumérable infini contient un sous-ensemble calculable infini.

PREUVE: Comme A est récursivement énumérable et infini, il existe une fonction f telle que $f(\mathbb{N}) = A$. Soit $g : \mathbb{N} \to \mathbb{N}$ définie par :

$$\begin{array}{rcl} g(0) & = & f(0) \\ g(n+1) & = & f(\mu z.f(z) > g(n)) \end{array}$$

La fonction g est totale car A est infini, et on a $g(n) < g(n+1)$. D'après l'exercice précédent, on en déduit que $g(\mathbb{N}) \subseteq A$ est un ensemble calculable. ∎

Exercice: 4.2.3

Montrez que l'ensemble des codes e tel que $\Phi_e(0) = 0$ n'est pas calculable.

PREUVE: Soit a le code de la fonction récursive partielle telle que :

$$\begin{array}{rcll} \Phi_a(e,n) & = & 0 & \text{si } \Phi_e(e)\downarrow \\ \Phi_a(e,n) & = & \uparrow & \text{sinon} \end{array}$$

On a que

$$\Phi_{S_1^1(a,e)}(n) = \Phi_a(e,n)$$

Soit $f(e) = S_1^1(a,e)$. On a donc :

$$\Phi_{f(e)}(n) = \Phi_a(e,n)$$

Et on a donc :

$$\begin{array}{rcll} \Phi_{f(e)}(0) & = & 0 & \text{si } \Phi_e(e)\downarrow \\ \Phi_{f(e)}(0) & = & \uparrow & \text{sinon} \end{array}$$

Soit A l'ensemble des codes e telles que $\Phi_e(0) = 0$. On a donc que $e \in \emptyset'$ ssi $f(e) \in A$. Donc $\emptyset' \leq_m A$, ce qui implique que A n'est pas calculable. ■

Exercice: 4.2.4

Montrez que l'ensemble des codes e tel que $\Phi_e(0)\downarrow \neq \Phi_e(1)\downarrow$ n'est pas calculable.

PREUVE: Soit a le code de la fonction récursive partielle telle que :

$$\begin{array}{rcll} \Phi_a(e,n) & = & n & \text{si } \Phi_e(e)\downarrow \\ \Phi_a(e,n) & = & \uparrow & \text{sinon} \end{array}$$

En utilisant le théorème SMN on a donc une fonction primitive récursive f telle que :

$$\begin{array}{rcll} \Phi_{f(e)}(n) & = & n & \text{si } \Phi_e(e)\downarrow \\ \Phi_{f(e)}(n) & = & \uparrow & \text{sinon} \end{array}$$

Soit A l'ensemble des codes e telles que $\Phi_e(0)\downarrow \neq \Phi_e(1)\downarrow$. On a donc que $e \in \emptyset'$ ssi $f(e) \in A$. Donc $\emptyset' \leq_m A$, ce qui implique que A n'est pas calculable. ■

Exercice: 4.2.5

Montrez que l'ensemble des codes e tel que $\forall n\ \Phi_e(n)\downarrow$ n'est pas calculable.

PREUVE: On peut utiliser exactement la même fonction f que celle de l'exercice précédent, avec la même preuve. ■

Exercice: 4.2.6

Soit $A = \{e \ : \ \exists n \ \Phi_e(n)\uparrow\}$. Soit $B = \{e \ : \ \exists n \text{ pair } \ \Phi_e(n)\uparrow\}$. Montrez que $A \equiv_m B$.

PREUVE: Soit a le code de la fonction récursive partielle telle que :

$$\begin{array}{rcll} \Phi_a(e,n) & = & \Phi_e(n/2) & \text{si } n \text{ est pair} \\ & = & \Phi_e(n) & \text{sinon} \end{array}$$

En utilisant le théorème SMN on a donc une fonction primitive récursive f telle que :

$$\begin{array}{rcll} \Phi_{f(e)}(n) & = & \Phi_e(n/2) & \text{si } n \text{ est pair} \\ & = & \Phi_e(n) & \text{sinon} \end{array}$$

Supposons $e \in B$. Alors clairement $e \in A$. Donc $B \leq_e A$. A l'inverse supposons $e \in A$. Alors on a $f(e) \in B$. Donc $A \leq_e B$. ∎

Exercice: 4.3.4

Montrez qu'un ensemble simple n'est pas calculable.

PREUVE: Soit A un ensemble simple. A intersecte tous les ensembles récursivement énumérables infinis. Comme $\omega - A$ est infini, et comme A n'intersecte pas $\omega - A$, on en déduit que $\omega - A$ n'est pas récursivement énumérable. Donc A n'est pas calculable. ∎

Exercice: 4.3.5

Montrez que $\emptyset'$ n'est pas un ensemble simple.

PREUVE: Soit a le code d'une fonction telle que $\Phi_a(e,m)\uparrow$. On a en particulier $\Phi_{S_1^1(e)}(m)\uparrow$ pour tout e et tout m. En particulier pour tout e on a $\Phi_{S_1^1(e)}(S_1^1(e))\uparrow$. Soit $A = \{S_1^1(e) \ : \ e \in \mathbb{N}\}$. On a clairement que A est récursivement énumérable. Pourtant $A \cap \emptyset'$ est vide. ∎

Exercice: 4.3.6

Montrez qu'il existe un ensemble récursivement énumérable W tel que si $\exists n \in W_e$ avec $n \geq 2e$, alors $W \cap W_e \neq \varnothing$, et tel que pour tout e, l'ensemble W contient au plus e éléments inférieur à $2e$. En déduire qu'il existe un ensemble récursivement énumérable simple.

PREUVE: L'ensemble W est énuméré de la manière suivante : A l'étape de calcul $t+1$, pour tout $e \leq t$ tel que $W_e[t] \cap W[t] = \varnothing$ et tel que $\exists n \geq 2e$ avec $n \in W_e[t]$, on énumère n dans W. Montrons que cette description informelle peut être décrite plus formellement. On définit la fonction primitive récursive $\Phi : \mathbb{N}^2 \to \mathbb{N}$ qui sera telle que $W = \{n \in \mathbb{N} \ : \ \exists e \ \exists t \ \Phi(e,t) = n+1\}$. On utilise la valeur 0 comme valeur spéciale : $\Phi(e,t) = 0$ doit être interprété comme "aucun élément de W_e n'est énuméré dans W à l'étape t". A l'inverse

$\Phi(e,t) = n > 0$ doit être interpretté comme "l'élément $n-1 \in W_e$ est énuméré dans W à l'étape t".

$$\begin{array}{rcll} \Phi(e,0) & = & 0 & \\ \Phi(e,t+1) & = & \min\{n \leq t \text{ tel que } P(e,t,n)\}+1 & \text{si } \exists n \leq t \text{ tel que } P(e,t,n) \\ & = & 0 & \text{sinon} \end{array}$$

avec $P(e,t,n)$ un raccourcis de notation pour :

$$n \geq 2e \text{ et } n \in W_e[t] \text{ et } e \leq t \text{ et } \forall m \in W_e[t]\ \forall s \leq t \text{ on a } \Phi(e,s) \neq m+1$$

On a bien que W est récursivement énumérable. Aussi les éléments de W plus petits que $2e$ ne peuvent être énumérés qu'en provenance d'ensembles W_a pour $a < e$. Par ailleurs il y a au plus un tel élément énuméré pour chaque W_a. On a donc au plus e éléments plus petit que $2e$ dans W. On en déduit que $\mathbb{N} - W$ est infini. Il est donc claire que W est un ensemble simple. ■

Exercice: 4.3.8

Deux ensemble A, B avec $A \cap B = \varnothing = 0$ sont récursivement séparables si il existe un ensemble calculable C tel que $A \subseteq C$ et $C \cap B = \varnothing$. Montrez qu'il existe deux ensembles récursivement énumérables A, B qui ne sont pas récursivement séparables.

Exercice: 4.3.9

Montrez qu'il existe deux ensembles calculables A, B tel que l'ensemble $C = \{x - y \ : x \in A \text{ et } y \in B\}$ est non-calculable.