Calculabilité et incomplétude - Notes de cours

Arnaud Durand, Paul Rozière
Paris7 – M2 LMFI

21 septembre 2021
(version provisoire — 18: 53)

Chapitre 1

Modèles de calcul

AVERTISSEMENT : version préliminaire des notes du cours fondamental 2, susceptible de corrections et d'ajouts.

L'objet de cette partie est de préciser les notions d'effectivité en mathématiques en proposant plusieurs modélisations de la notion de fonction et d'ensemble calculables. On va introduire essentiellement trois modèles différents, à savoir :

— la définition de Kleene des fonctions calculables : les fonctions μ-récursives ;
— les fonctions calculables par machines à registres ;
— les fonctions calculables par machines de Turing.

On montre ensuite que ces trois approches de la calculabilité (il en existe bien d'autres) qui mettent en jeux, a priori, des notions de ressources différentes mènent à des notions équivalentes de fonction calculable. Les notes se poursuivent ensuite par une introduction aux résultats de base de la théorie de la calculabilité.

La première approche présentée est celle des fonctions μ-récursives. Les objets de référence sont les fonctions sur les entiers naturels. On considère dans ce cadre qu'une fonction est calculable si elle appartient à un certain ensemble de base (fonctions constantes, successeur, ...) ou si elle peut être obtenue à partir de l'ensemble de fonctions de bases par composition, définition par récurrence ou d'autres schémas que l'on détaillera. Ces fonctions sont intuitivement calculables, mais, contrairement aux définitions que l'on verra ensuite, le calcul reste implicite. Cette approche élégante, fournit un intermédiaire commode pour montrer que les diverses notions de fonctions calculables sont équivalentes. Mais la formalisation du calcul s'avère très vite indispensable pour prolonger l'étude. Il s'avère que n'importe quel modèle de calcul convient et conduit aux mêmes résultats, pourvu qu'il soit suffisamment riche.

Notation. Dans la suite, un vecteur de paramètres sera alternativement désigné par $(x_1, ..., x_p)$ ou par $\overline{x}$ suivant les situations. On parle d'*arité* pour désigner le nombre de paramètres des fonctions et relations.

1.1 Fonctions récursives primitives

Les fonctions récursives primitives sont essentiellement les fonctions qui se calculent par récurrence sur un argument entier, et les composées de celles-ci. Elles ont été introduites dans les années 1920, et les mathématiciens se sont rendu compte assez vite qu'elles ne pouvaient représenter toutes les fonctions calculables (Ackermann, 1926), même si « beaucoup » de fonctions calculables « usuelles » sur les entiers sont récursives primitives.

Définition 1.1.1 L'ensemble des *fonctions récursives primitives* est le plus petit sous-ensemble de l'ensemble des fonctions à plusieurs arguments entiers à valeurs entières ($\bigcup_{p\in\mathbb{N}^*} \mathbb{N}^{\mathbb{N}^p}$) qui satisfait les conditions suivantes :

i. il contient la fonction *nulle* $\lambda x.0 : \mathbb{N} \to \mathbb{N}$, la fonction *successeur* $\mathrm{s} : \mathbb{N} \to \mathbb{N}$ et les *projections* p_i^k : $\mathbb{N}^k \to \mathbb{N}$ $(1 \leq i \leq k)$, définies par $p_k^i(x_1, \ldots, x_k) = x_i$;

ii. il est clos par le *schéma de composition* :
si $h : \mathbb{N}^p \to \mathbb{N}$, et $g_1, \ldots, g_p : \mathbb{N}^n \to \mathbb{N}$ sont récursives primitives, alors $f : \mathbb{N}^n \to \mathbb{N}$ définie par $f(x_1, \ldots, x_n) = h(g_1(x_1, \ldots, x_n), \ldots, g_p(x_1, \ldots, x_n))$ est récursive primitive.

iii. il est clos par le *schéma de récurrence primitive* :
si $g : \mathbb{N}^p \to \mathbb{N}$ et $h : \mathbb{N}^{p+2} \to \mathbb{N}$ sont récursives primitives, alors $f : \mathbb{N}^{p+1} \to \mathbb{N}$ est récursive primitive, f définie par :

$$f(a_1, \ldots, a_p, 0) = g(a_1, \ldots, a_p)$$
$$f(a_1, \ldots, a_p, x+1) = h(a_1, \ldots, a_p, x, f(a_1, \ldots, a_p, x)) .$$

On parlera de *définition récursive primitive* d'une fonction, pour une définition de la fonction qui utilise ces trois clauses.

Un prédicat P sur $\mathbb{N}^p$ (resp. un sous-ensemble E de $\mathbb{N}^p$) est un *prédicat récursif primitif* (resp. *un sous-ensemble récursif primitif*), quand sa fonction caractéristique est récursive primitive.

On rappelle que la fonction caractéristique d'un ensemble est définie par $\chi_A(\overline{x}) = 1$ si $\overline{x} \in A$ et $\chi_A(\overline{x}) = 0$ sinon ; la fonction caractéristique d'un prédicat est celle de l'ensemble des uples pour lesquels le prédicat est vrai.

Plus généralement On dira d'un sous-ensemble de $\bigcup_{p\in\mathbb{N}^*} \mathbb{N}^{\mathbb{N}^p}$ qu'il est *clos par opérations récursives primitives* s'il satisfait les trois clauses **i**, **ii** et **iii** ci-dessus, sans être nécessairement le plus petit. Intuitivement, l'ensemble de toutes les fonctions calculables, doit être clos par opérations récursives primitives. Cela sera démontré quand nous aurons une caractérisation satisfaisante de la notion de fonction calculable.

1.1.1 Exemples de fonctions récursives primitives

Fonctions constantes Pour $n \in \mathbb{N}$, on note s^n la fonction successeur composée n fois. La fonction constante $c_n : \mathbb{N} \to \mathbb{N}$ telle que $c_n(x) = n$ se définit par $c_n(x) = \mathrm{s}^n(\lambda x.0(x))$, en utilisant donc n fois le schéma de composition.

Addition, Multiplication, exponentielle Une définition par récurrence de l'addition est :

$$x + 0 = x \text{ et } x + (y+1) = (x+y) + 1 .$$

et cette définition est récursive primitive. De façon plus explicite, la fonction $+ : \mathbb{N}^2 \to \mathbb{N}$ est définie par

$$+(x, 0) = p^1_1(x) \text{ et } +(x, y+1) = \mathrm{s}(p^3_3(x, y, +(x, y))) .$$

L'addition est bien obtenue à partir des fonctions initiales p^1_1, p^3_1, p^3_3, s, d'une occurrence du schéma de composition, et d'une occurrence du schéma de récurrence primitive.

De même la fonction multiplication $\times : \mathbb{N}^2 \to \mathbb{N}$, définie par récurrence par

$$x \cdot 0 = 0 \text{ et } x \cdot (y+1) = x + x \cdot y$$

est aussi récursive primitive :

$$\times(x, 0) = 0 \text{ et } \times(x, y+1) = +(p^3_1(x, y, +(x, y)), p^3_3(x, y, \times(x, y)))$$

ainsi que la fonction exponentielle définie par

$$x^0 = 0 \text{ et } x^{y+1} = x^y \cdot x .$$

Définitions par récurrence primitive de fonctions à un argument Le schéma de récurrence primitive donné en définition ne permet pas de définir directement des fonctions à un argument. Cependant, on montre facilement qu'il s'étend par :

iii$_0$ Si $b \in \mathbb{N}$, $h : \mathbb{N}^2 \to \mathbb{N}$ est récursive primitive, alors f est récursive primitive, $f : \mathbb{N} \to \mathbb{N}$ définie par

$$f(0) = b$$
$$f(x+1) = h(x, f(x)) .$$

En effet il suffit de définir par récurrence primitive une fonction auxiliaire $aux : \mathbb{N}^2 \to \mathbb{N}$, avec un second argument inutile :

$$aux(a,0) = c_b(a) \text{ et } aux(a,x+1) = h(p_2^3(a,x,f(x)), p_1^3(a,x,f(x))) \; ; f(x) = aux(p_1^1(x), p_1^1(x)).$$

Par exemple la fonction factorielle définie par

$$0! = 1 \text{ et } (x+1)! = (x+1)\cdot x!$$

est récursive primitive en suivant **iii$_0$**. Ce schéma est utilisé au paragraphe suivant pour les fonctions de signe et le prédécesseur.

Signe, prédécesseur, soustraction Les deux fonctions de signe sg et $\overline{\text{sg}}$ qui suivent sont récursives primitives

$$\text{sg}(0) = 0 \text{ et } \text{sg}(x+1) = 1 \; ; \; \overline{\text{sg}}(0) = 1 \text{ et } \overline{\text{sg}}(x+1) = 0$$

($\text{sg}(x+1) = c_1(p_1^2(x, \text{sg}(x))$).
Les fonctions récursives primitives sont partout définies. On définit un prédécesseur nul en 0 et une soustraction tronquée :

$$\text{pred}(0) = 0 \text{ et } \text{pred}(x+1) = x \; ; \; x \dot{-} 0 = x \text{ et } x \dot{-} (y+1) = \text{pred}(x \dot{-} y).$$

avec $\text{pred}(0) = 0$ et $\text{pred}(x+1) = p_1^2(x, \text{pred}(a,x))$.

Dès que l'on a une définition par récurrence sur un argument à partir de fonctions récursives primitives, la fonction obtenue est récursive primitive. Cette restriction aux récurrences à un argument ne peut être levée en toute généralité : on verra qu'il existe des fonctions comme la fonction d'Ackermann (voir 1.2.2 page 13) qui se définissent par récurrence double et ne sont pas récursives primitives. Ainsi une définition naturelle de la fonction $\dot{-}$ par récurrence double est

$$\dot{-}(x+1, y+1) = \dot{-}(x,y), \; \dot{-}(x,0) = x \text{ et } \dot{-}(0,y) = 0 \, .$$

cette définition induit manifestement un calcul, d'ailleurs plutôt plus naturel et plus efficace, de la fonction $\dot{-}$, mais elle ne met pas en évidence que cette fonction est récursive primitive (voir cependant l'exercice 8 page 11).

Comparaison Les prédicats de comparaison $\leq, \geq, <, >, =, \neq$ sont récursifs primitifs, c'est-à-dire que leurs fonctions caractérisques le sont. En effet : $\chi_>(x,y) = \text{sg}(x \dot{-} y)$, $\chi_\leq(x,y) = \overline{\text{sg}}(x \dot{-} y)$, et donc $\chi_=(x,y) = \overline{\text{sg}}(x \dot{-} y) \cdot \overline{\text{sg}}(y \dot{-} x)$, $\chi_\neq(x,y) = \overline{\text{sg}}(\chi_=(x,y))$.

1.1.2 Propriétés de clôtures

Au delà de la définition, l'ensemble des fonctions récursives primitives, et plus généralement tout ensemble clos par opérations récursives primitives, satisfait un grand nombre de propriétés de clôture. On a déjà vu la définition par récurrence primitive des fonctions à un argument. On en détaille quelques autres ci-dessous.

Définition par itération On peut ne pas utiliser tous les arguments dans la récurrence. Par exemple l'addition et la multiplication utilisent le schéma de définition par itération qui est de la forme $f(\overline{x}, 0) = g(\overline{x})$ et $f(\overline{x}, y+1) = h(\overline{x}, f(\overline{x}, y))$. On montre facilement que les fonctions définies ainsi sont récursives primitives.

Sommes et produits bornés Si f de $\mathbb{N}^{p+1} \to \mathbb{N}$ est une fonction récursive primitive, les fonctions g et h de $\mathbb{N}^{p+1} \to \mathbb{N}$ définies par

$$g(x_1, \ldots, x_p, y) = \sum_{i=0}^{y} f(x_1, \ldots, x_p, i) \quad \text{et} \quad h(x_1, \ldots, x_p, y) = \prod_{i=0}^{y} f(x_1, \ldots, x_p, i)$$

sont récursives primitives. En effet (pour la somme) :

$$g(\overline{x}, 0) = f(\overline{x}, 0) \text{ et } g(\overline{x}, y+1) = f(\overline{x}, y+1) + g(\overline{x}, y).$$

(Le schéma de récurrence utilisé est celui de la définition, ce n'est pas une définition par itération).

Opérations logiques L'ensemble des prédicats récursifs primitifs d'arité quelconque est clos par opération booléennes (conjonction, disjonction, négation). Ainsi, si A et B sont des prédicats récursifs primitifs, alors $P(\overline{x},\overline{y}) \wedge Q(\overline{x},\overline{z})$ (pensez à $\chi_P(\overline{x},\overline{y}) \cdot \chi_Q(\overline{x},\overline{z})$) est récursif primitif. La fonction $\overline{\text{sg}}$ permet d'obtenir la négation, et donc la disjonction.

Opérations ensemblistes Les résultats précédents se traduisent immédiatement de façon ensembliste : la classe des sous-ensembles récursifs primitifs de $\mathbb{N}^p$, p fixé, est close par intersection, réunion et passage au complémentaire. La classe des ensembles récursifs primitifs est close par produit cartésien.

Définition par cas Soient f et g deux fonctions récursives primitives et A un ensemble récursif primitif alors la fonction h suivante est récursive primitive :

$$\begin{array}{ll} h(\overline{x}) & = f(\overline{x}) \text{ si } \overline{x} \in A \\ h(\overline{x}) & = g(\overline{x}) \text{ sinon.} \end{array}$$

Cela se voit simplement en remarquant que $h(\overline{x}) = f(\overline{x}) \cdot \chi_A(\overline{x}) + f(\overline{x}) \cdot \chi_{\mathbb{N}^k \setminus A}(\overline{x})$. Plus généralement, si $A_1, ..., A_n$ sont des ensembles récursifs primitifs deux à deux disjoints et $f_1, ..., f_n, g$ des fonctions récursives primitives alors la fonction h suivante est récursive primitive :

$$\begin{array}{ll} h(\overline{x}) & = f_1(\overline{x}) \text{ si } \overline{x} \in A_1 \\ h(\overline{x}) & = f_2(\overline{x}) \text{ si } \overline{x} \in A_2 \\ \vdots & \vdots \\ h(\overline{x}) & = f_n(\overline{x}) \text{ si } \overline{x} \in A_n \\ h(\overline{x}) & = g(\overline{x}) \text{ si } \overline{x} \notin A_1 \cup \cdots \cup A_n. \end{array}$$

Minimisation bornée Soit h une fonction récursive primitive. Une fonction f est obtenue par schéma de *minimisation bornée* à partir de h si elle est définie par :

$$\begin{array}{lcll} f(x_1,\ldots,x_p,y) & = & \text{le plus petit entier } t \leq y \text{ tel que } h(x_1,\ldots,x_p,t) = 0 & \text{s'il existe un tel entier,} \\ f(x_1,\ldots,x_p,y) & = & 0 \text{ s'il n'existe pas de tel entier.} & \end{array}$$

On note l'opération de minimisation bornée :

$$f(\overline{x},y) = \mu t \leq y.[h(\overline{x},t) = 0] \ .$$

La fonction f ainsi obtenue est récursive primitive. En effet :

$$\begin{array}{lcl} f(\overline{x},0) & = & 0 \\ f(\overline{x},y+1) & = & \text{sg}\big(h(\overline{x},0)\big) \cdot \big(f(\overline{x},y) + \overline{\text{sg}}\big(f(\overline{x},y)\big) \cdot \overline{\text{sg}}\big(h(\overline{x},y+1)\big) \cdot (y+1)\big) \end{array}$$

Par composition, si k est récursive primitive, alors f définie par $f(\overline{x}) = \mu t \leq k(\overline{x}).[h(\overline{x},t) = 0]$ est aussi récursive primitive.

Quantifications bornées Si P est un prédicat récursif primitif alors les deux prédicats P_e et P_q définis comme suit sont récursifs primitifs :

$$\begin{array}{l} P_e\, x_1 \ldots x_p\, y \equiv \exists z \leq y\, P\, x_1 \ldots x_p\, y \\ P_q\, x_1 \ldots x_p\, y \equiv \forall z \leq y\, P\, x_1 \ldots x_p\, y \ . \end{array}$$

En effet

$$\begin{array}{l} \chi_{P_e}(\overline{x},y) = \text{sg}(\sum_{z=0}^{y} \chi_P(\overline{x},z)) \\ \chi_{P_q}(\overline{x},y) = \prod_{z=0}^{y} \chi_P(\overline{x},z) \qquad . \end{array}$$

Par composition avec les fonctions de projections, les prédicats (dépendants des mêmes variables que P)

$$\begin{array}{l} \exists z \leq x_i\, P\, x_1 \ldots x_p \\ \forall z \leq x_i\, P\, x_1 \ldots x_p \ . \end{array}$$

sont aussi récursifs primitifs. Plus généralement, on montre par composition que les prédicats

$$\exists z \leq f(\overline{x})\ P\overline{x}$$
$$\forall z \leq f(\overline{x})\ P\overline{x}$$

sont récursifs primitifs dès que le quantificateur est borné par une fonction récursive primitive f.

Les propriétés de clôture de ce paragraphe se généralisent évidemment à tout ensemble de fonctions arithmétiques clos par opérations récursives primitives, puisque seul cette partie de la définition des fonctions récursives primitives a été utilisée.

Exercice 1 Montrer que les sous-ensembles finis et cofinis des $\mathbb{N}^k$, $k \in \mathbb{N}$, sont récursifs primitifs.

Exercice 2 On a vu que l'ensemble des prédicats récursifs primitifs d'arité quelconque est clos sous les opérations booléennes (conjonction, disjonction, négation), (voir page 6). Détailler la démonstration et en déduire que la classe des ensembles récursifs primitifs est close par réunion, intersection, produit cartésien et passage au complémentaire.

1.1.3 Prédicats définissables au premier ordre par quantification bornée

On considère ici un sous-ensemble de prédicats récursifs primitifs qui contient la plupart des prédicats arithmétiques naturels. Appelons $\mathscr{R}$ le plus petit ensemble contenant les prédicats d'addition, de multiplication et clos par opérations booléennes et quantification bornée par un polynôme. En d'autres termes un prédicat $R(x_1, ..., x_k)$ est dans $\mathscr{R}$ s'il est définissable par une formule du premier ordre sur la signature $\{+, \times\}$ et dont toutes les quantifications sont bornées par un polynôme en $x_1, ..., x_k$.

Tout prédicat de $\mathscr{R}$ est récursif primitif (au vu de ce que l'on a déjà prouvé). La classe $\mathscr{R}$ nous fournit un moyen simple (par une définition logique) de montrer que certains prédicats sont récursifs primitifs. Par exemple, un nombre p est premier s'il satisfait :

$$p \text{ est premier } \equiv p \geq 2 \wedge \forall x \leq p\, \forall y \leq p\ \big(x \cdot y = p \rightarrow \big(x = 1 \vee x = p\big)\big)\ .$$

De même, x divise y, noté $x|y$ s'écrit : $x|y \equiv \exists z \leq y\ x \cdot z = y$. De façon générale, la plupart des prédicats arithmétiques naturels sont dans $\mathscr{R}$.

Enfin, en utilisant la clôture par minimisation bornée et par récurrence primitive on montre à partir de l'exemple précédent que la fonction $\mathrm{p} : \mathbb{N} \rightarrow \mathbb{N}$ qui à n associe $\mathrm{p}(n)$ (noté p_n) le $n+1$-ème nombre premier est bien récursive primitive ($\mathrm{p}_0 = 2$, $\mathrm{p}_1 = 3$, ...). Il suffit de remarquer que le $n+1$-ème nombre premier est forcément borné par $\mathrm{p}_n! + 1$ (la factorielle est récursive primitive). La définition de p se fait alors par récurrence :

$$\mathrm{p}(0) = 2 \text{ et } \mathrm{p}(n+1) = \mu x \leq \mathrm{p}(n)! + 1.[x \text{ est premier } \wedge\ x > \mathrm{p}(n)].$$

Exercice 3 (division euclidienne) Montrer que que le prédicat de divisibilité $|$ est récursif primitif, et que les fonctions quotient : $\mathbb{N}^2 \rightarrow \mathbb{N}$ (quotient de la division de n par p), reste : $\mathbb{N}^2 \rightarrow \mathbb{N}$ (reste de la division de n par p) sont des fonctions récursives primitives.

1.1.4 Premiers codages

Les paramètres des fonctions récursives primitives sont des entiers, et ce sera encore le cas pour les diverses notions de fonctions calculables que nous allons étudier. Il est cependant bien évident que la notion de calcul dépasse de loin ce cadre restreint. Dans un sens, un modèle de calcul général doit pouvoir prendre en entrée des objets finis de natures très différentes : nombres autres qu'entiers, mots sur un alphabet fini, structures algébriques finies, graphes, hypergraphes, arbres,... La notion de calcul a intuitivement un sens dans tous ces contextes. De même, clairement, les fonctions calculables doivent pouvoir, à travers une représentation finie (les programmes qui les calculent) être considérées comme des paramètres valides d'autres fonctions calculables.

Dans cette section, on va montrer comment représenter à l'aide d'entiers (on dira souvent « coder »), tout d'abord les couples et n-uplets, puis les listes — c'est-à-dire les suites finies — d'entiers. Ces codages se manipulent par des fonctions récursives primitives, c'est-à-dire que les fonctions usuelles,

par exemple sur les listes (longueur, i-ème élément, sous-liste, etc) sont représentées par des fonctions récursives primitives. De plus ces codages permettent d'établir de nouvelles propriétés de clôture pour les fonctions récursives primitives, définition par récurrence en fonction d'un entier strictement plus petit (qui n'est pas nécessairement le prédécesseur) par exemple. La méthode utilisée pour les listes se généralise à des structures plus complexes comme les arbres étiquetés, ce que l'on verra dans la partie 3.2.

La bijection de Cantor

On appelle α la bijection de Cantor $\mathbb{N}\times\mathbb{N}$ dans $\mathbb{N}$, définie par :

$$\alpha(n,p) = (\sum_{i=0}^{n+p} i) + p = \frac{(n+p+1)(n+p)}{2} + p$$

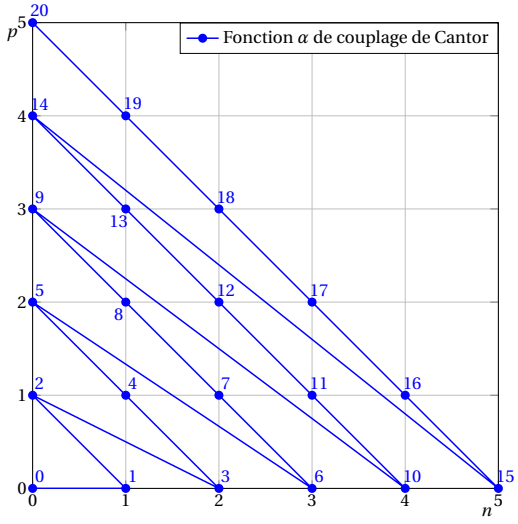
On vérifie facilement que α est bijective, strictement croissante pour ses deux entrées n et p et qu'elle est récursive primitive. Ona également $n \le \alpha(n,p)$ et $p \le \alpha(n,p)$.

L'application étant bijective, on peut définir deux projections π_1 et π_2 vérifiant pour tout entier c et tout couple d'entiers (n,p) :

$$\begin{aligned}&\alpha(\pi_1(c),\pi_2(c)) = c,\\ &\pi_1(\alpha(n,p)) = n\\ &\pi_2(\alpha(n,p)) = p\end{aligned}$$

qui sont également récursives primitives. En effet :

$$\begin{aligned}&\pi_1(x) = \mu z \le x.[\exists t \le x\ \alpha(z,t) = x]\\ &\pi_2(x) = \mu t \le x.[\exists z \le x\ \alpha(z,t) = x].\end{aligned}$$



On peut alors définir par récurrence sur $k \ge 1$ les fonctions $\alpha_k : \mathbb{N}^k \to \mathbb{N}$ par :

$$\begin{aligned}&\alpha_1(n) = n\\ &\alpha_{k+1}(n_1,\dots,n_{k+1}) = \alpha_2(n_1,\alpha_k(n_2,\dots,n_{k+1}))\quad (\text{ en particulier } \alpha_2 = \alpha)\ .\end{aligned}$$

À nouveau on démontre par récurrence sur k que chaque α_k, $k \in \mathbb{N}^*$, est une bijection. Les projections correspondantes, notées π_i^k pour $1 \le i \le k$ sont définies par :

$$\text{pour } 1 \le i < k,\ \pi_i^k = \pi_1 \circ \underbrace{\pi_2 \circ \cdots \circ \pi_2}_{i-1}\ ;\quad \pi_k^k = \underbrace{\pi_2 \circ \cdots \circ \pi_2}_{k-1}\quad (\text{en particulier } \pi_1^2 = \pi_1 \text{ et } \pi_2^2 = \pi_2).$$

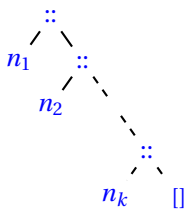
Chacune des fonctions α_k, de même que les projections π_i^k, sont récursives primitives par composition. On pose $\alpha_k(x_1,\dots,x_k) = \langle x_1,\dots,x_k\rangle$ et on utilisera le plus souvent cette dernière écriture.

Cette famille de fonctions ne permet pas, telle quelle, de définir une bijection de l'ensemble des suites finies vers $\mathbb{N}$ puisque précisement chacune de ces fonctions est bijective. Par exemple : $\alpha_3(1,0,0) = \alpha_2(1,\alpha_2(0,0)) = \alpha_2(1,0)$. On va modifier un tout petit peu l'approche pour obtenir un codage bijectif des suites finies, les listes de l'informatique.

Un codage bijectif des suites finies

Pour coder les suites finies d'entiers de taille arbitraire, ou listes d'entiers, on utilise à nouveau le codage des couples.

Le principe du codage est illustré par le schéma ci-contre. Le code de la liste vide (noté []) est 0, le code d'une liste non vide est donné par le couple constitué de premier élément de la liste et du code du reste de la liste, et ce couple doit être non nul.



La fonction cons : $\mathbb{N}^2 \to \mathbb{N}$ associe à x et y l'entier noté $x :: y$, elle est définie par :

$$x :: y = 1 + \alpha(x, y)$$

On obtient ainsi une fonction récursive primitive bijective de $\mathbb{N}^2 \to \mathbb{N}^*$. On appelle hd (pour *head*) et tl (pour *tail*) les fonctions vérifiant :

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

La fonction liste de l'ensemble $\mathscr{S}$ des suites finies d'entiers dans $\mathbb{N}$ est définie inductivemement, on note $[a_0; \ldots; a_n] = \mathrm{liste}((a_0, \ldots, a_n))$:

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

La fonction liste est bijective : on démontre par récurrence sur l'entier l que celui-ci possède un unique antécédent, en utilisant que α_2 est bijective. La fonction :: est récursive primitive (car α_2 l'est). Les fonctions hd et tl sont récursives primitives :

$$\mathrm{hd} = \pi_1 \circ \mathrm{pred} \quad \text{et} \quad \mathrm{tl} = \pi_2 \circ \mathrm{pred}\ .$$

La fonction nthl qui à l et i associe la suite codée par l à partie du $i+1$-ème élément (0 sinon), et la fonction nth qui à l et i associe le $i+1$-ème élément de la suite codée par l (0 sinon), sont récursives primitives :

$$\begin{array}{lll} \mathrm{nthl}(l, 0) = l & \text{puis} & \mathrm{nth}(l, 0) = 0 \\ \mathrm{nthl}(l, i+1) = \mathrm{tl}(\mathrm{nthl}(l, i)) & & \mathrm{nth}(l, i+1) = \mathrm{hd}(\mathrm{nthl}(l, i))\ . \end{array}$$

Ce codage bijectif des listes se généralise facilement à d'autres types de données inductifs [1], par exemple celui des arbres binaires étiquetés de la section 3.2.1 page 65 (utilisé pour la démonstration du premier théorème d'incomplétude).

Récurrence primitive sur la suite des valeurs On peut souhaiter faire dépendre la récurrence non seulement de la dernière valeur obtenue pour la fonction f mais de tout (ou partie) des précédentes. On parle de *récurrence sur la suite des valeurs*. L'ensemble des fonctions récursives primitives est clos par le schéma de récurrence sur la suite des valeurs. Même si elle fait appel au codage des listes introduit ci-dessus, la proposition suivante est pour une large part indépendante du codage choisi.

Lemme 1.1.2 *soient $g : \mathbb{N}^p \to \mathbb{N}$ et $h : \mathbb{N}^{p+2} \to \mathbb{N}$ deux fonctions récursives primitives, alors la fonction $f : \mathbb{N}^{p+1} \to \mathbb{N}$ suivante est récursive primitive :*

$$\begin{array}{l} f(a_1, \ldots, a_p, 0) = g(a_1, \ldots, a_p) \\ f(a_1, \ldots, a_p, x+1) = h\big(a_1, \ldots, a_p, x, \big[f(a_1, \ldots, a_p, x); \ldots; f(a_1, \ldots, a_p, 0)\big]\big)\ . \end{array}$$

L'appel fait appel à la suite des précédents, donc peut faire appel à n'importe quel d'entre eux avec la fonction nth du paragraphe précédente.

Démonstration. Appelons $\tilde{f}(\overline{a}, x) = \big[f(\overline{a}, x); \ldots; f(\overline{a}, 0)\big]$. On montre tout d'abord que $\tilde{f}$ est récursive primitive. En effet, $\tilde{f}(\overline{a}, 0) = g(\overline{a}) :: 0$ et

$$\tilde{f}(\overline{a}, x+1) = f(\overline{a}, x+1) :: \tilde{f}(\overline{a}, x) = h\big(\overline{a}, x, \tilde{f}(\overline{a}, x)\big) :: \tilde{f}(\overline{a}, x)$$

On a alors $f(\overline{a}, x) = \mathrm{hd}(\tilde{f}(\overline{a}, x))$. ■

1. Ces codages sont empruntés à KOMARA et VODA 1999

On laisse en exercice le fait de montrer que les fonctions suivantes (dont on se servira dans la suite) sont récursives primitives :

— la fonction mem fonction caractéristique de l'appartenance d'un entier n à une suite codée par l,
— la fonction @ vérifiant que $l@l'$ est le code de la concaténation des suites codées par l et l',
— la fonction len qui à un entier l associe la longueur de la suite codée par l,
— la fonction inc : $\mathbb{N}^2 \to \mathbb{N}$ qui à i et $l = [a_0;\ldots;a_i;\ldots;a_n]$ associe $\text{inc}(i,l) = [a_0;\ldots;a_i+1;\ldots;a_n]$ quand $i \leq n$, $\text{inc}(i,l) = l$ sinon.

Exercice 4 (Codage des listes par décomposition en nombres premiers) Un autre codage des listes s'appuie sur la décomposition en nombres premiers. On note $\mathscr{S}$ l'ensemble des suites finies d'entiers. Soit $\text{p} : \mathbb{N} \to \mathbb{N}$ la fonction qui à n associe le $n+1$-ème nombre premier noté p_n. Cette fonction est récursive primitive. La fonction de codage des listes $\text{seq} : \mathscr{S} \to \mathbb{N}$ associe à chaque suite de longueur finie $(x_1,\ldots,x_k)$ la valeur suivante

$$\text{seq}(x_1,...,x_k) = p_0^k \cdot p_1^{x_1} \cdot p_2^{x_2} \cdots p_k^{x_k},$$

avec la convention pour la suite vide (), seq() = 1.

1. Montrer que ce codage est injectif mais pas surjectif (il construit des nombres très grands ce qui le rendrait difficilement utilisable en pratique).

2. Montrer que la fonction de $\mathbb{N}^2 \to \mathbb{N}$ qui à (x,n) associe l'exposant de p_n dans la décomposition en facteurs premiers de x est récursive primitive.

3. En déduire que

3.a. il existe une fonction récursive primitive qui calcule le n-ième élément d'une suite représentée par x, quand x représente une suite de longueur supérieure ou égale à n (on ne se préoccupe pas de la valeur de la fonction dans les autres cas) ;

3.b. il existe une fonction récursive primitive qui calcule la longueur de la suite codée par x, quand x représente une suite ;

3.c. La fonction caractéristique de l'ensemble des codes de suites (l'ensemble image de la fonction seq) est récursive primitive.

4. Montrer qu'il existe une fonction récursive primitive qui, à deux entiers $\text{seq}(x_1,...,x_k)$ et $\text{seq}(y_1,...,y_h)$ codant des suites renvoie le nombre représentant la concaténation des deux listes $\text{seq}(x_1,...,x_k,y_1,...,y_h)$.

Exercice 5 (Définition par récurrences mutuelles) Utiliser la fonction α_k pour montrer que si les fonctions $g_1,\ldots,g_k : \mathbb{N}^n \to \mathbb{N}$, et $h_1,\ldots,h_k : \mathbb{N}^{n+k+1} \to \mathbb{N}$ sont récursives primitives, alors, les fonctions $f_1,\ldots,f_k$ définies ci-dessous sont récursives primitives (on écrit $\overline{a}$ pour $a_1,\ldots,a_n$).

$$\begin{array}{ll} f_1(\overline{a},0) = g_1(\overline{a}) & f_k(\overline{a},0) = g_k(\overline{a}) \\ f_1(\overline{a},x+1) = h_1(\overline{a},x,f_1(\overline{a},x),\ldots,f_k(\overline{a},x)) & \cdots \quad f_k(\overline{a},x+1) = h_k(\overline{a},x,f_1(\overline{a},x),\ldots,f_k(\overline{a},x)) \end{array}$$

Exercice 6 (récurrence sur les listes) Le but de l'exercice est de montrer que la définition par récurrence naturelle sur la structure de liste se code de façon récursive primitive.

1. Montrer que l'ensemble des fonctions récursives primitives est clos par le schéma de récurrence sur les listes (préciser pourquoi f est bien définie) : si $g : \mathbb{N}^p \to \mathbb{N}$ et $h : \mathbb{N}^{p+3} \to \mathbb{N}$ sont récursives primitives, alors $f : \mathbb{N}^{p+1} \to \mathbb{N}$ est récursive primitive, f définie par

$$\begin{array}{l} f(a_1,\ldots,a_p,[\,]) = g(a_1,\ldots,a_p) \\ f(a_1,\ldots,a_p,x::l) = h\big(a_1,\ldots,a_p,x,l,f(a_1,\ldots,a_p,l)\big)\,. \end{array}$$

et l'utiliser pour montrer que la fonction mem fonction caractéristique de l'appartenance d'un entier n à une suite codée par l, la fonction @ vérifiant que $l@l'$ est le code de la concaténation des suites codées par l et l', la fonction len qui à un entier l associe la longueur de la suite codée par l, sont récursives primitives.

2. Montrer que si $f : \mathbb{N}^{p+1} \to \mathbb{N}$ est récursive primitive, alors la fonction map_f qui à $a_1, \ldots, a_p, l$ codant $(u_i)_{i\leq n}$ associe $\text{map}_f(l)$ codant $\big(f(a_1, \ldots, a_p, u_i)\big)_{i\leq n}$ est récursive primitive.

Montrer que la fonction $concat$ qui à un entier l codant une liste de liste $\big((u_i)_{i\leq n_i}\big)_{i\leq p}$ associe l'entier $concat(l)$ codant la suite des entiers de chaque suite $(u_i)_{i\leq n_i}$ dans le même ordre est récursive primitive.

Montrer que la fonction $subst$ qui à trois entiers l, k, v, associe le code la suite obtenue en remplaçant dans la suite codée par l toutes les occurrences de v par les entiers de la suite codée par k est récursive primitive (on peut se servir des deux fonctions précédentes).

Exercice 7 (récurrence avec substitution de paramètre) On appelle *schéma de récurrence avec substitution de paramètre* le schéma de récurrence suivant (énoncé ici avec un seul paramètre) qui étant données g, $\gamma : \mathbb{N} \to \mathbb{N}$ et $h : \mathbb{N}^3 \to \mathbb{N}$ définit $f : \mathbb{N}^2 \to \mathbb{N}$.

$$\begin{aligned} f(a,0) &= g(a) \\ f(a,x+1) &= h(a,x,f(\gamma(a),x)) . \end{aligned}$$

Intuitivement, ce schéma conserve le fait d'être calculable (le calcul termine puisque la variable de récurrence décroit). La récurrence avec substitution de paramètre apparaît naturellement avec des structures de données plus complexes que les entiers, par exemple quand on définit une fonction par récurrence sur la longueur d'une liste (le paramètre est la liste elle-même), ou sur la hauteur d'un arbre (le paramètre est l'arbre).

On montre que l'ensemble des fonctions récursives primitives est clos sous ce schéma. On suppose dans la suite g, γ et h récursives primitives, et f définie comme ci-dessus. On note $\gamma^p(x) = \underbrace{\gamma \circ \cdots \circ \gamma}_{p}(x)$ (γ^0 est l'identité).

1. Montrer que la fonction F définie ci-dessous est récursive primitive

$$\begin{aligned} F(p,a,0) &= g\left(\gamma^p(a)\right) \\ F(p,a,x+1) &= h\left(\gamma^{p\dot{-}(x+1)}(a), x, F(p,a,x)\right) . \end{aligned}$$

2. Montrer que :

$$\forall x,a,p \in \mathbb{N}\ \left(x \leq p \to F(p,a,x) = f(\gamma^{p\dot{-}x}(a),x)\right)$$

et en déduire que f est récursive primitive.

3. Application : montrer que la fonction $\text{inc} : \mathbb{N}^2 \to \mathbb{N}$ qui à i et $l = [a_0;\ldots;a_i;\ldots;a_n]$ associe $\text{inc}(i,l) = [a_0;\ldots;a_i+1;\ldots;a_n]$ quand $i \leq n$, $\text{inc}(i,l) = l$ sinon, est récursive primitive.

Exercice 8 (récurrence double sans imbrication) On verra que la récurrence double ne conserve pas en général le fait d'être récursif primitif (la fonction d'Ackermann, voir 1.2.2 page 13, est définie par récurrence double). On peut montrer que s'il n'y a pas *imbrication* des appels récursifs dans la récurrence double, alors celle-ci reste « récursive primitive ».

On suppose que a et b sont des entiers, que h est une fonction récursive primitive à 4 arguments, que h_2 est une fonction primitive récursive à 2 arguments.

Pour simplifier les notations, les schémas qui suivent sont donnés sans paramètres, les démonstrations étant essentiellement les mêmes en présence de paramètres.

1. Montrer que la fonction f définie par :

$$\begin{aligned} f(0,y) &= a , \\ f(x+1,0) &= b , \\ f(x+1,y+1) &= h(x,y,f(x,y),f(x+1,y)) . \end{aligned}$$

est récursive primitive (on peut utiliser le codage des couples et la récurrence sur la suite des valeurs).

2. (généralisation, plus difficile) Montrer que la fonction f définie par :

$$\begin{aligned} f(0,y) &= a, \\ f(x+1,0) &= b, \\ f(x+1,y+1) &= h(x,y,f(x,h_2(x,y)),f(x+1,y)). \end{aligned}$$

est récursive primitive. On peut procéder comme à la question précédente, mais en cherchant pour h_2 donné une nouvelle fonction α' de codage des couples (injective mais non nécessairement bijective) vérifiant :

$$\alpha'(x+1,y) < \alpha'(x+1,y+1) \text{ et } \alpha'(x,h_2(x,y)) < \alpha'(x+1,y+1).$$

Rózsa Péter a étudié à partir des années 1930 de façon assez systématique la clôture de l'ensemble des fonctions récursives primitives sous divers schémas de récurrence dont ceux exposés ci-dessus [2].

1.2 Au delà des fonctions récursives primitives

1.2.1 Évaluation des fonctions récursives primitives

Dans ce paragraphe, « fonction calculable » signifie « fonction calculable au sens intuitif » : on dispose d'un procédé qui pour une entrée donnée permet d'obtenir en un temps fini le résultat de la fonction appliquée à cette entrée. On va voir que les fonctions récursives primitives ne résument pas à elles seules l'ensemble des fonctions calculables au sens intuitif, pour une raison qui tient au fait même que les fonctions récursives primitives sont calculables.

Pour s'en persuader, admettons tout d'abord qu'il est possible de *coder* les définitions des fonctions récursives primitives par des entiers, en utilisant par exemple les codages des suites (voir la section partie 1.1.4), ces définitions étant des suites finies de lettres, de façon analogue aux codages que l'on réalisera pour les machines en section 1.3 (ou mieux, ceux que l'on verra en section 3.2). Il est alors assez simple de se rendre compte que la fonction caractéristique de l'ensemble de ces codes est calculable (et même récursive primitive, sauf choix de codage aberrant), ce qui correspond à l'idée intuitive que l'on sait reconnaître si un assemblage de lettres est bien la définition d'une fonction récursive primitive. De la même façon on admet que l'arité de la fonction de code i se calcule en fonction de i (la fonction étant même récursive primitive, sauf, à nouveau, choix de codage aberrant).

La fonction d'évaluation des fonctions récursives primitives à un argument est alors une fonction Eval à deux arguments, telle que, si i est le code d'une fonction f récursive primitive à un argument, alors pour tout entier x, $\mathsf{Eval}(i,x) = f(x)$. La fonction peut être prolongée de façon arbitraire, par exemple elle vaut toujours 0, quand i n'est pas un code de fonction récursive primitive à un argument.

Dire des fonctions récursives primitives qu'elles sont calculables, c'est bien dire que cette fonction d'évaluation est calculable. Or cette fonction ne peut être récursive primitive. On le montre par diagonalisation : si elle l'était, la fonction à un argument $x \mapsto \mathsf{Eval}(x,x)+1$ le serait. Or si cette fonction était récursive primitive, elle aurait un code n, et on aurait $\mathsf{Eval}(n,n)+1 = \mathsf{Eval}(n,n)$.

On peut essayer d'analyser la définition de la fonction Eval, pour observer à quel endroit elle ne rentre pas dans le cadre des fonctions récursives primitives. Réaliser les codages qui permettent d'écrire en détail la fonction Eval est un peu long mais ne présente pas de difficulté particulière, et on verra ultérieurement plusieurs exemples de tels codages. Contentons nous de définir une syntaxe pour les fonctions récursives primitives, et de décrire l'évaluation pour cette syntaxe.

Les termes récursifs primitifs d'arité n ($n \in \mathbb{N}^*$) sont définis inductivement, sachant que même si on reprend pour simplifier les mêmes notations que pour les fonctions, un tel terme n'est qu'une suite de lettres :

i. les lettres $\lambda x.0$ et s sont des termes récursifs primitifs d'arité 1 ; pour $k \in \mathbb{N}^*$, $1 \le i \le k$, p_i^k est un terme récursif primitif d'arité k ;

ii. si h est un terme récursif primitif d'arité p et $g_1,\dots,g_p$ sont des termes récursifs primitifs d'arité n, alors $h \circ (g_1,\dots,g_p)$ est un terme récursif primitif d'arité n ;

2. voir PÉTER 1967

iii. si g et h sont des termes récursifs primitifs, g d'arité p et h d'arité $p+2$, alors $\mathrm{rec}(g,h)$ est un terme récursif primitif d'arité $p+1$.

L'interprétation de ces termes par des fonctions récursives primitives est celle que l'on imagine, la famille de fonctions (Eval_n) permet de l'expliciter (on retrouve la fonction Eval par $\mathsf{Eval} = \mathsf{Eval}_1$ et, grâce au codages de uples, il n'est pas difficile de définir (Eval_n) à partir de Eval_1).

i. $\mathsf{Eval}_1(\lambda x.0, x) = 0$, $\mathsf{Eval}_1(\mathrm{s}, x) = x+1$, $\mathsf{Eval}_k\left(p_i^k, x_1, \ldots, x_k\right) = x_i$;

ii. $\mathsf{Eval}_n\left(h \circ (g_1, \ldots, g_p), x_1, \ldots, x_n\right) = \mathsf{Eval}_p\left(h, \mathsf{Eval}_n(g_1, x_1, \ldots, x_n), \ldots, \mathsf{Eval}_n(g_p, x_1, \ldots, x_n)\right)$;

iii. $\mathsf{Eval}_{p+1}\left(\mathrm{rec}(g,h), a_1, \ldots, a_p, 0\right) = \mathsf{Eval}_p\left(g, a_1, \ldots, a_p\right)$
$\mathsf{Eval}_{p+1}\left(\mathrm{rec}(g,h), a_1, \ldots, a_p, x+1\right) = \mathsf{Eval}_{p+2}\left(h, a_1, \ldots, a_p, x, \mathsf{Eval}_{p+1}\left(\mathrm{rec}(g,h), a_1, \ldots, a_p, x\right)\right)$.

On suppose donc que les termes récursifs primitifs sont codés par des entiers, et que de plus un entier codant un terme est strictement supérieur aux entiers codant ses sous-termes stricts, ce qui est le cas pour les choix naturels de codage.

On peut voir alors la définition des fonctions (Eval_n) comme une définition par récurrence mutuelle, l'argument de récurrence, celui qui décroît, est le code de la fonction. En examinant celle-ci on s'aperçoit que

— l'évaluation des fonctions initiales est récursive primitive : ce sont de simples compositions des fonctions initiales ;
— La condition donnée par l'évaluation de la composition reste dans le cadre récursif primitif : c'est une généralisation de la définition par récurrences mutuelles (exercice 5 page 10) ;
— l'évaluation de la récurrence primitive utilise une récurrence double, avec imbrication des appels aux fonctions Eval : c'est cette clause de la définition de l'évaluation qui fait que celle-ci n'est pas récursive primitive.

1.2.2 Fonction d'Ackermann

La méthode qui précède est très générale, mais lourde à mettre en œuvre explicitement. Elle peut se simplifier, non plus en énumérant toutes les fonctions récursives primitives, mais en énumérant une suite Ack_n de fonctions récursives primitives d'arité 1, de façon que Ack_n soit supérieure à partir d'un certain rang à toutes les fonctions récursives primitives utilisant moins de n occurrences du schéma de récurrence [3]. La fonction à deux arguments définie par $\mathrm{Ack}(n,x) = \mathrm{Ack}_n(x)$ énumère les fonctions Ack_n, et la fonction diagonale $x \mapsto \mathrm{Ack}(x,x)$ sera alors supérieure à partir d'un certain rang à toute fonction récursive primitive, donc ne pourra être récursive primitive.

Cette fonction est appelée *fonction d'Ackermann* [4]. Elle se définit par récurrence double avec imbrication des appels récursifs :

$$\begin{aligned}
&\mathrm{Ack}(0,x) = x+2\\
&\mathrm{Ack}(1,0) = 0\\
&\mathrm{Ack}(n+2,0) = 1\\
&\mathrm{Ack}(n+1,x+1) = \mathrm{Ack}(n,\mathrm{Ack}(n+1,x))
\end{aligned}$$

Chaque fonction $\mathrm{Ack}_n : x \mapsto \mathrm{Ack}(n,x)$ est bien récursive primitive : la fonction Ack_{n+1} est obtenue en itérant la fonction Ack_n

$$\mathrm{Ack}_{n+1}(x+1) = \mathrm{Ack}_n(\mathrm{Ack}_{n+1}(x)) .$$

Toutes les fonctions Ack_n sont croissantes, et cette croissance est de plus en plus rapide

$$\mathrm{Ack}_1(x) = 2x,\ \mathrm{Ack}_2(x) = 2^x,\ \mathrm{Ack}_3(x) = 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}\}x},\ \ldots$$

On dit qu'une fonction $f : \mathbb{N} \longrightarrow \mathbb{N}$ *domine* une fonction $g : \mathbb{N}^p \longrightarrow \mathbb{N}$ si f est supérieure à g à partir d'un certain rang, i.e.

$$\exists K \in \mathbb{N}\ \forall \vec{x} \in \mathbb{N}^p\ g(\overline{x}) \leq f(\sup(\overline{x}, K))$$

3. La fonction Ack_n est même supérieure à partir d'un certain rang aux fonctions récursives primitives utilisant au plus n occurrences *imbriquées* du schéma de récurrence, voir la définition des $\mathscr{C}_n$ page 14.

4. La fonction originale avait 3 arguments, Rozsa Péter a donné une version a 2 arguments à laquelle celle donnée ici est quasi-identique).

Alors on peut montrer que pour toute fonction primitive récursive f, il existe n tel que Ack_n domine f, ce dont on déduit

Proposition 1.2.1 *La fonction d'Ackermann n'est pas récursive primitive.*

Les détails sont donnés dans la section suivante et en exercice.

Une hiérarchie des fonctions récursives primitives On définit une suite $\mathscr{C}_n$ d'ensembles de fonctions primitives récursives tous clos par composition, les fonctions de $\mathscr{C}_n$ étant les fonctions qui utilisent des suites imbriquées d'au plus n schémas de récurrence primitive. En voici une définition par induction :

(i) $\mathscr{C}_0$ est la clôture par composition de l'ensemble des fonctions de bases : constantes, projections, et successeur ;

(ii) $\mathscr{C}_{n+1}$ est la clôture par composition de la réunion de $\mathscr{C}_n$ et des fonctions obtenues par une seule occurrence du schéma de récurrence primitive à partir des fonctions de $\mathscr{C}_n$.

Il est clair que $\mathscr{C}_n \subset \mathscr{C}_{n+1}$ et que $\bigcup_{i=0}^{\infty} \mathscr{C}_n$ est l'ensemble de toutes les fonctions récursives primitives. On vérifie facilement que $\mathrm{Ack}_n \in \mathscr{C}_n$. On peut maintenant préciser l'énoncé du paragraphe précédent.

Proposition 1.2.2 *Si $f \in \mathscr{C}_n$, alors Ack_{n+1} domine f, en particulier $\mathrm{Ack}_{n+1} \notin \mathscr{C}_n$.*

La hiérarchie des $\mathscr{C}_n$ est donc stricte, puisque $\mathrm{Ack}_{n+1} \in \mathscr{C}_{n+1}$, mais $\mathrm{Ack}_{n+1} \notin \mathscr{C}_n$.

Remarque. En particulier, les fonctions de $\mathscr{C}_2$, niveau 3 de la hiérarchie, sont connues sous le nom de *fonctions élémentaires au sens de Kalmar*. La fonction Ack_3 n'est pas élémentaire.

Les démonstrations, en particulier celle de la proposition précédente, sont détaillées dans l'exercice qui suit.

Exercice 9 (fonction d'Ackermann)

1. Vérifier qu'il existe bien une et une seule fonction de $\mathbb{N}^2 \to \mathbb{N}$ vérifiant les équations de la fonction d'Ackermann. Calculez explicitement les premières valeurs de Ack, par exemple $\{\mathrm{Ack}(n,x) \mid 0 \le n \le 3,\ 0 \le x \le 3\}$, et donner un argument informel pour la calculabilité (au sens intuitif) de Ack, c'est-à-dire la raison pour laquelle la suite des appels récursifs termine (indication : utiliser l'ordre lexicographique sur les couples).

2. Montrer que
$$\forall n \in \mathbb{N}\ \forall x > 0\ \mathrm{Ack}_{n+1}(x) = \underbrace{\mathrm{Ack}_n \circ \cdots \circ \mathrm{Ack}_n}_{x}(\mathrm{Ack}_{n+1}(0))$$
et vérifier les expressions des fonctions Ack_1, Ack_2 et Ack_3 données ci-dessus.

3. Vérifiez que chacune des fonctions Ack_n est récursive primitive, et donnez en une définition dont vous montrerez qu'elle utilise exactement n instances du schéma de définition par itération vu en section 1.1.2 page 5. On peut remarquer que les n schémas de récurrences sont imbriqués.

4. Montrer que $\forall n \in \mathbb{N} \forall x \in \mathbb{N}^*\ \mathrm{Ack}_n(x) > x$.

5. En déduire que pour tout entier n, Ack_n est strictement croissante.

6. Déduire de la question 4 que, à partir de 2, Ack est croissante au sens large sur son premier argument, le second étant fixé :
$$\forall x \ge 2\ \forall n \in \mathbb{N}\ \mathrm{Ack}(n,x) \le \mathrm{Ack}(n+1,x)\,.$$

La hiérarchie $\mathscr{C}_n$ des fonctions récursives primitives, ainsi que la définition de $f : \mathbb{N} \longrightarrow \mathbb{N}$ *domine* $g : \mathbb{N}^p \longrightarrow \mathbb{N}$ ont été données ci-dessus.
On pose pour k entier, $\mathrm{Ack}_n^k = \underbrace{\mathrm{Ack}_n \circ \cdots \circ \mathrm{Ack}_n}_{k}$.

7. Montrer que : $\forall n, k \in \mathbb{N}\ \mathrm{Ack}_n^k \in \mathscr{C}_n$

8. Montrer que $\forall n, k, x \in \mathbb{N}\ \mathrm{Ack}_n^k(x) \le \mathrm{Ack}_{n+1}(x+k)\,.$

9. Montrer par récurrence sur la définition de l'ensemble des fonctions récursives primitives que :

$$\text{si } f \in \mathscr{C}_n, \text{ alors } \exists k \in \mathbb{N}\ \mathrm{Ack}_n^k \text{ domine } f.$$

10. Montrer que Ack_n^k est dominée par Ack_{n+1} (on pourra montrer que pour $y > 0$, $\mathrm{Ack}_{n+1}(y) \geq 2y$, puis que pour $x > 2k$, $\mathrm{Ack}_{n+1}(x-k) \geq x$).

11. En déduire que si $f \in \mathscr{C}_n$, alors Ack_{n+1} domine f.

12. En déduire que la fonction d'Ackermann n'est pas récursive primitive.
On peut également montrer que la fonction diagonale $n \mapsto \mathrm{Ack}(n,n)$ domine toutes les fonctions récursives primitives.

1.2.3 Fonctions partielles μ-récursives

La méthode décrite (informellement) pour montrer que la fonction d'évaluation des fonctions récursives primitives n'est pas récursive primitive est très générale. Elle permet de montrer par diagonalisation que la fonction d'évaluation de n'importe quel ensemble de fonctions calculables n'appartient pas à cet ensemble, pourvu essentiellement que deux conditions soient réalisées.

1. La première que l'on a déjà indiquée, c'est que l'on puisse reconnaître de façon effective les codes des fonctions de l'ensemble en question.

2. La seconde qui est restée implicite jusqu'à présent, c'est que ces fonctions soient *partout définies*. En effet la contradiction n'arrive que si la fonction Eval est définie en (n,n) (ce qui est forcément le cas pour la fonction d'évaluation des fonctions récursives primitives).

Il est possible de remettre en cause la première condition. Mais le plus naturel du point de vue du calcul est de remettre en cause la seconde condition, c'est-à-dire d'étendre la notion de fonction calculable aux fonctions partielles. L'intuition est qu'une fonction partielle f n'est pas définie en une valeur x donnée, quand le calcul de $f(x)$ se poursuit indéfiniment sans jamais rendre une valeur. Dans les langages de programmation usuels cela peut arriver dès que l'on utilise une boucle `while`.

L'introduction de fonctions partielles calculables date des années 1930, bien avant les débuts de l'informatique. Il se trouve qu'un programme dont l'exécution ne termine pas est une chose indispensable en informatique. Par exemple une boucle interactive, qui attend une intervention de l'utilisateur n'a pas à terminer (sauf sur ordre de l'utilisateur). A fortiori un système d'exploitation ne termine pas sauf intervention extérieure. Cependant, même si l'activité des ordinateurs ne se résume pas, très loin de là, au calcul de fonctions, c'est la seule chose que formalise la calculabilité et la seule chose dont nous nous préoccuperons ici.

Le schéma de minimisation On va donc d'une part considérer des fonctions partielles de $\mathbb{N}^k$ dans $\mathbb{N}$ c'est à dire des fonctions définies sur un sous ensemble A de $\mathbb{N}^k$ (et non définies en dehors), d'autre part introduire un nouveau schéma de définition, qui peut produire des fonctions partielles à partir de fonctions totales. Il sera également nécessaire de généraliser la composition et la récurrence primitive aux fonctions partielles.

Notation : on écrira $f(x)\downarrow$ pour f est définie en x, $f(x)\uparrow$ pour f n'est pas définie en x.

Définition 1.2.3 L'opérateur μ est défini par le schéma suivant dit schéma μ ou schéma de minimisation. Soit f une fonction partielle calculable :

$$z = \mu y.\big(f(\overline{x},y) = 0\big) \quad \text{ssi} \begin{cases} f(\overline{x},z) = 0 \\ \text{pour tout } y < z, f(\overline{x},y)\downarrow \text{ et } f(\overline{x},y) \neq 0. \end{cases}$$

Le schéma de minimisation ci-dessus introduit une fonction de $\overline{x}$ qui est a priori partielle, même si la fonction f est totale (récursive primitive par exemple). On trouvera forcément des fonctions calculables f telles que que pour un certain uple $\overline{x}$, $f(\overline{x},y) \neq 0$ pour toute valeur de y, et alors $\mu y.\big(f(\overline{x},y) = 0\big)$ n'est simplement pas défini.

Clairement, le schéma de minimisation doit être défini de telle sorte que si $\mu y.\big(f(\overline{x},y) = 0\big)$ a une valeur z alors on peut obtenir ce z par un calcul. C'est bien le cas ici : trouver z, revient à calculer

successivement $f(\overline{x},0)$, $f(\overline{x},1)$, ... et à s'arrêter au premier y tel que $f(\overline{x},y)=0$. Vu ainsi, on a bien besoin que chacune des valeurs intermédiaires soit définie pour que le calcul arrive à son terme.

Imaginons la version suivante du schéma de minimisation :

$$z = \min y.\big(f(\overline{x},y)=0\big) \quad \text{ssi} \begin{cases} f(\overline{x},z)=0 \\ \text{pour tout } y<z, f(\overline{x},y)\neq 0. \end{cases}$$

Si z est le plus petit entier tel que $f(\overline{x},z)=0$ mais qu'il existe $y<z$ tel que $f(\overline{x},y)$ est non définie, il n'existe pas de procédure évidente pour calculer z. De fait, on montrera que ce schéma ne peut convenir quand on aura une caractérisation satisfaisante de fonction partielle calculable (exercice 17 page 42).

Si on souhaite obtenir des fonctions totales à l'issue d'une étape de minimisation, il faut imposer de n'appliquer ce schéma qu'à des fonctions totales calculables f telles que :

$$\forall \overline{x}\, \exists y\ f(\overline{x},y)=0.$$

Une telle fonction calculable f sera dite *régulière*.

Le schéma de minimisation conserve le fait d'être « calculable » (en un sens intuitif), pour les fonctions partielles comme totales. Il s'avère, qu'en ajoutant le schéma de minimisation aux schémas de clôtures des fonctions récursives primitives, on obtient en fait une caractérisation satisfaisante de la notion de fonction calculable (cela sera précisé à la section 1.5.3).

On appelle pour l'instant *fonctions μ-récursives* les fonctions calculables au sens de cette définition introduite par Kleene. Plus tard on les appelera simplement fonctions calculables.

Définition 1.2.4 (fonction partielle μ-récursive) L'ensemble des *fonctions partielles μ-récursives* est le plus petit sous-ensemble de fonctions partielles à plusieurs arguments entiers

i. contenant la fonction nulle, les projections et la fonction successeur

ii. clos par composition des fonctions partielles si $h:\mathbb{N}^p \longrightarrow \mathbb{N}$, et $g_1,\ldots,g_p:\mathbb{N}^n \longrightarrow \mathbb{N}$ sont des fonctions partielles μ-récursives, alors $f:\mathbb{N}^n \longrightarrow \mathbb{N}$ définie ci-dessous est une fonction partielle μ-récursive.

$$\begin{array}{l} \text{si } g_1(x_1,\ldots,x_n)\downarrow \ldots g_p(x_1,\ldots,x_n)\downarrow \text{ et } h(g_1(x_1,\ldots,x_n),\ldots,g_p(x_1,\ldots,x_n))\downarrow \\ \quad \text{alors } f(x_1,\ldots,x_n)\downarrow \text{ et } f(x_1,\ldots,x_n)=h(g_1(x_1,\ldots,x_n),\ldots,g_p(x_1,\ldots,x_n)) \\ \text{sinon } f(x_1,\ldots,x_n)\uparrow \end{array}$$

iii. clos par récurrence primitive pour les fonctions partielles si $g:\mathbb{N}^p \longrightarrow \mathbb{N}$ et $h:\mathbb{N}^{p+2} \longrightarrow \mathbb{N}$ sont des fonctions partielles μ-récursives, alors $f:\mathbb{N}^{p+1} \longrightarrow \mathbb{N}$ définie ci-dessous est une fonction partielle μ-récursive.

$$\begin{array}{l} \text{si } g(a_1,\ldots,a_p)\downarrow \text{ alors } f(a_1,\ldots,a_p,0)\downarrow \text{ et } f(a_1,\ldots,a_p,0)=g(a_1,\ldots,a_p) \\ \text{si } f(a_1,\ldots,a_p,x)\downarrow \text{ et } h(a_1,\ldots,a_p,x,f(a_1,\ldots,a_p,x))\downarrow \\ \quad \text{alors } f(a_1,\ldots,a_p,x+1)\downarrow \text{ et } f(a_1,\ldots,a_p,x+1)=h(a_1,\ldots,a_p,x,f(a_1,\ldots,a_p,x)) \\ \text{sinon } f(a_1,\ldots,a_p,x+1)\uparrow \end{array}$$

iv. clos par le schéma de minimisation μ : si $g:\mathbb{N}^{p+1} \longrightarrow \mathbb{N}$ est une fonction partielle μ-récursive, alors la fonction partielle $f:\mathbb{N}^p \longrightarrow \mathbb{N}$ définie ci-dessous notée $f(x_1,\ldots,x_p)=\mu z.g(x_1,\ldots,x_p,z)=0$ est μ-récursive.

$$\begin{array}{l} \text{si } \exists y_0\ \big[g(x_1,\ldots,x_p,y_0)=0 \wedge \forall z<y_0 (g(x_1,\ldots,x_p,z)\downarrow \wedge g(x_1,\ldots,x_p,z)\neq 0)\big] \\ \quad \text{alors } f(x_1,\ldots,x_p)\downarrow \text{ et } f(x_1,\ldots,x_p)=y_0 \\ \quad \text{sinon } f(x_1,\ldots,x_p)\uparrow \end{array}$$

Par exemple la fonction nulle part définie $x \mapsto \mu t.0=1$ est partielle μ-récursive, de même la fonction $x \mapsto \mu t.x=0$, définie seulement en 0.

En appliquant le schéma de minimisation aux fonctions régulières seulement, on évite de parler de fonction partielle.

Définition 1.2.5 (fonction totale μ-récursive) L'ensemble des *fonctions totales μ-récursives* est le plus petit ensemble de fonctions (partout définies) à plusieurs arguments entiers clos par opérations récursives primitives et schéma de minimisation.

Malgré les apparences, cette définition est très différente de celle des fonctions récursives primitives ou de celles des fonctions partielles μ-récursives. En effet, comme il ne s'agit que de fonctions totales, la condition de régularité doit être vérifiée pour chaque utilisation du schéma de minimisation. On peut donc donner une définition plus explicite.

Fait 1.2.6 *L'ensemble des fonctions totales μ-récursives est le plus petit sous-ensemble de $\mathscr{F}$*
- *clos par opérations récursives primitives (voir page 4) ;*
- *tel que si f récursive totale est μ-récursive, f d'arité $k+1$ et f régulière, alors $\overline{x} \mapsto \mu y \cdot \big(f(\overline{x}, y) = 0\big)$, qui est totale, est μ-récursive.*

Or la condition de régularité n'est pas, contrairement aux autres, une condition « syntaxique ». En fait elle ne peut-être vérifiable mécaniquement. On le verra de façon précise ultérieurement, mais l'argument est celui développé en début de cette section : la première des deux conditions de la page 15 ne peut être vérifiée.

Clairement toute fonction totale μ-récursive est une fonction partielle μ-récursive qui est totale. Par composition on peut définir des fonctions partielles μ-récursives qui s'avèrent totales à partir de fonctions qui ne le sont pas, comme par exemple en composant une fonction définie seulement en 0 et la fonction nulle. La réciproque n'est donc pas aussi évidente. Elle sera obtenue par codage du calcul à la section suivante (voir proposition 2.1.2 page 33), ce qui lèvera les ambiguïtés que pourrait susciter ces définitions. Jusqu'à cette proposition, on distingue entre fonction totale μ-récursive au sens de la définition 1.2.5, et fonction partielle μ-récursive (au sens donc de la définition 1.2.4) qui s'avère totale.

Définition 1.2.7 (prédicat décidable) Un prédicat d'arité p sur les entiers, un sous-ensemble de $\mathbb{N}^p$, est dit *décidable* ou *calculable* quand sa fonction caractéristique est calculable, c'est-à-dire μ-récursive avec la définition dont on dispose actuellement (noter qu'une fonction caractéristique est toujours totale).

L'ensemble des fonctions totales μ-récursives étant clos par opérations récursives primitives, non seulement contient les fonctions récursives primitives, mais a également les propriétés de clôture de la section 1.1.2 page 5. Par conséquent l'ensemble des prédicats décidables a aussi les propriétés de clôture de la section 1.1.2 vues pour les prédicats récursifs primitifs.

De façon analogue aux fonctions régulières, on appelle *prédicat régulier* un prédicat P d'arité $p+1$ calculable vérifiant

$$\forall \overline{x} \exists y P\, \overline{x}\, y.$$

et l'ensemble des fonction totales μ-récursives est clos par schéma de minimisation pour les prédicats réguliers, c'est à dire que si $P\,\overline{x}\,y$ est régulier, la fonction f qui à $\overline{x}$ associe le plus petit y vérifiant $P\overline{x}\,y$, notée $\overline{x} \mapsto \mu y \cdot P\,\overline{x}\,y$, qui est totale, est μ-récursive.

On aimerait disposer pour les fonctions partielles calculables de propriétés et schémas de clôture analogues à ceux obtenus pour les fonctions totales calculables (μ-récursives). Une contrainte importante au sujet des fonctions partielles calculables est que dans une composition $f \circ g$ n'est définie en x que si $g(x)$ est définie. Par exemple $x \mapsto f(x) - f(x)$ n'est définie que si f est définie. C'est l'appel par valeurs des langages de programmation fonctionnels. Or cela n'est pas toujours ce qui est attendu. Ainsi on souhaite la propriété de clôture suivante.

Proposition 1.2.8 *Si P est un prédicat décidable d'arité p, et f et g deux fonctions partielles d'arité p μ-récursives, alors la fonction partielle h définie ci-dessous est μ-récursive partielle*

$$\textit{si } P\,\overline{x} \textit{ alors } h(\overline{x}) = f(\overline{x}) \textit{ sinon } h(\overline{x}) = g(\overline{x})\ .$$

Si f et g sont totales on a vu que $h(\overline{x}) = \chi_P(\overline{x})\,f(\overline{x}) + \overline{\mathrm{sg}}(\chi_P(\overline{x}))\,g(\overline{x})$ convient, mais si f et g sont partielles ce n'est plus le cas. En effet quand on a $P\,\overline{x}$ et $f\,\overline{x})\downarrow$, on souhaite que h soit définie indépendamment de

ce qui se passe pour g. Or ce n'est pas le cas pour la fonction h définie ci-dessus si $g(\overline{x})\uparrow$. Il n'y a pas a priori de façon simple de définir la fonction h qui doit vérifier

$$\text{si } P\,\overline{x} \text{ et } f(\overline{x})\downarrow \text{ alors } h(\overline{x})\downarrow\ ;\ \text{ si } \neg P\,\overline{x} \text{ et } g(\overline{x})\downarrow \text{ alors } h(\overline{x})\downarrow\ ;\ \text{ dans les autres cas } h(\overline{x})\uparrow .$$

Démontrer cette proposition à partir de la caractérisation des fonctions μ-récursives n'a rien d'évident. La démonstration se fera facilement après avoir montré l'équivalence avec une caractérisation des fonctions partielles calculables où le calcul est explicite (exercice 14 page 34).

En effet ce formalisme des a été introduit par Kleene non pour son expressivité, mais parce qu'il fournit les primitives pour coder n'importe quelle notion de calcul sur machine, comme on va le constater dans le cas particulier des machines à registres.

1.3 Fonctions calculables par machines à registres

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. Elles disposent d'autre part d'un programme qui est une suite finie d'instructions.

L'*état d'une machine* (à un instant donné) est le contenu des registres d'une part, un index de lecture du programme d'autre part. Cet index est un entier inférieur à la longueur du programme qui renvoie à l'instruction qui doit être exécutée : l'index de lecture est donc le numéro de cette instruction dans la suite finie constituant le programme (on choisit de numéroter de 1 en 1 à partir de 0). Chaque instruction agit sur l'état de la machine.

On va voir successivement plusieurs notions de programme, la première notion étant la plus primitive.

1.3.1 Programmes goto

Un *programme* goto est une suite finie constituée de 4 types d'instruction. à partir de 0.

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, passer à l'instruction suivante :
$$R_i := R_i - 1$$
(quand le registre est déjà à 0 il n'est pas modifié par l'instruction).
3. exécuter un goto conditionnel, si le registre numéro i est nul, aller à l'instruction numéro p (p est un entier inférieur à la longueur du programme) :
$$\mathsf{if}\ R_i = 0\ \mathsf{goto}\ p$$
4. une instruction d'arrêt qui apparait une et une seule fois en fin du programme :
$$\mathsf{halt}$$

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 la clarté. Le calcul d'une telle machine peut ne pas terminer, et l'instruction goto est la seule instruction susceptible de conduire le calcul à ne pas terminer.

Comme on peut accéder directement au registre numéro i : cela modélise plus ou moins la mémoire RAM (random access memory), mémoire à accès aléatoire. Vous verrez une autre notion de machine, les machines de Turing qui utilisent une mémoire à accès séquentiel : une machine de Turing écrit sur un ruban, il faut $|i-j|$ étapes pour aller de la case i à la case j.

Ces machines restent toutefois très théoriques : elles possèdent un nombre fini de registres qui n'est pas borné, de même que la taille de chaque registre, et la taille du programme.

Il manque par ailleurs des instructions essentielles pour manipuler les adresses de registres, même si elles ne permettraient pas de calculer de nouvelles fonctions (et ici le nombre fixé de registres rend ses instructions inutiles).

Définition 1.3.1 Une fonction partielle f de $\mathbb{N}^n \to \mathbb{N}$ est calculable par une machine M à k registres, signifie que quand on initialise la machine en affectant les entiers $x_1, \ldots, x_{\inf(n,k)}$ aux registres $R_1, \ldots, R_{\inf(n,k)}$, et la valeur 0 aux registres restant (s'il en existe), l'index de lecture étant à 0 (sur la première instruction), alors la machine termine son calcul si et seulement si $f(x_1, \ldots, x_n)\downarrow$, et dans ce cas dans le registre R_0 a la valeur $f(x_1, \ldots, x_n)$ à la fin du calcul.

On n'a pas supposé ci-dessus dans la définition que $n \leq k$, mais évidemment :

Lemme 1.3.2 *Si $f : \mathbb{N}^n \to \mathbb{N}$ est calculable par une machine à k registres, alors f est calculable par une machine à k registres avec $k \geq n+1$.*

Démonstration. En effet, si $k < n+1$, il suffit d'ajouter les registres manquant $R_k, \ldots, R_n$, qui ne seront pas modifiés lors du calcul. ■

Voyons quelques exemples.

La machine à un seul registre R_0 et dont le programme contient pour seule instruction halt calcule les fonctions constantes égales à 0, : $x_1 \ldots x_n \mapsto 0$.

$$\boxed{R_0} \quad 0 \quad halt$$

On calcule l'addition avec la machine à 4 registres dont le programme est le suivant :

$\boxed{R_0}\ \boxed{R_1}\ \boxed{R_2}\ \boxed{R_3}$

0 if $R_1 = 0$ goto 4
1 $R_1 := R_1 - 1$
2 $R_0 := R_0 + 1$
3 if $R_3 = 0$ goto 0
4 if $R_2 = 0$ goto 8
5 $R_2 := R_2 - 1$
6 $R_0 := R_0 + 1$
7 if $R_3 = 0$ goto 4
8 halt

Le registre R_3 ne sert qu'à pouvoir écrire un goto inconditionnel, instruction que l'on pourrait donc employer, sachant qu'on peut la simuler en ajoutant un registre à la machine. On va dans un premier temps montrer que l'on peut ainsi simuler quelques nouvelles instructions utiles.

Exercice 10

1. Décrire des machines qui calculent les fonctions sg et $\overline{\text{sg}}$.
2. Décrire une machine dont le calcul termine en 0 sur 0, et qui ne termine pas pour tout autre entier.

1.3.2 De nouvelles instructions

Proposition 1.3.3 *Si une fonction partielle de $\mathbb{N}^n \to \mathbb{N}$ est calculée par une machine à registres utilisant en plus des instructions usuelles l'une des instructions suivantes*

1. *le* goto *inconditionnel, aller à la ligne p :*

   goto p

2. *le registre numéro i est mis à* 0 *et l'on passe à l'instruction suivante :*

   $R_i := 0$

3. *l'assignation d'un entier, le registre numéro i reçoit le contenu du registre numéro j $(j \neq i)$ et l'on passe à l'instruction suivante :*
$$R_i := R_j$$

alors elle est calculable par une machine à registres usuelle.

Démonstration. On construit à chaque fois une machine avec programme goto qui simule une machine avec programme enrichi avec la nouvelle instruction.

1. goto p : on a vu qu'il suffisait d'ajouter un registre dont le contenu sera toujours nul, on suppose $n \leq k$ (Lemme 1.3.2), on ajoute alors un registre R_{k+1} et l'instruction devient if $R_{k+1} = 0$ goto p.
2. $R_i := 0$: la machine conserve les mêmes registres. On suppose que l'instruction $R_i := 0$ est à la place l. On remplace l'instruction $R_i := 0$ par la suite d'instructions :
$$\begin{array}{ll} l & \text{if } R_i = 0 \text{ goto } l+3 \\ l+1 & R_i := R_i - 1 \\ l+2 & \text{goto } l \end{array}$$
et dans le reste du programme on décale de 2 tous les goto l' pour $l' > l$ (remplacés par goto $l'+2$).
3. $R_i := R_j$: on suppose $n \leq k$ (Lemme 1.3.2), on ajoute alors un registre R_{k+1}. On suppose que l'instruction $R_i := R_j$ est à la place l. On remplace l'instruction $R_i := 0$ par la suite d'instructions :
$$\begin{array}{ll} l & R_i := 0 \\ l+1 & \text{if } R_j = 0 \text{ goto } l+6 \\ l+2 & R_j := R_j - 1 \\ l+3 & R_i := R_i + 1 \\ l+4 & R_{k+1} := R_{k+1} + 1 \\ l+5 & \text{goto } l+1 \\ l+6 & \text{if } R_{k+1} = 0 \text{ goto } l+10 \\ l+7 & R_{k+1} := R_{k+1} - 1 \\ l+8 & R_j := R_j + 1 \\ l+9 & \text{goto } l+6 \end{array}$$
et dans le reste du programme on décale de 9 les goto l' pour $l' > l$. ■

Par exemple, avec ces nouvelles instructions, la projection p_i^n est calculée par une machine à n registres de programme
$$\begin{array}{l} R_0 := R_i \\ \text{halt} \end{array}$$
la fonction successeur est calculée par la machine à 2 registres de programme :
$$\begin{array}{l} R_0 := R_1 \\ R_0 := R_0 + 1 \\ \text{halt} \end{array}$$
Ces fonctions sont donc calculables par une machine à registres avec programme goto.

1.3.3 Programmes structurés

On peut préferer utiliser des instructions plus complexes qui permettent de structurer les programmes.

Définition 1.3.4 Un *programme structuré* est une suite d'instructions, chaque instruction pouvant être (définition inductive) :

1. l'une des instructions d'assignation déjà décrites
$$R_i := 0,\ R_i := R_i + 1,\ R_i := R_i - 1,\ R_i := R_j\ (j \neq i)$$

2. une séquence d'instructions, elles sont exécutées séquentiellement, puis le programme passe à l'instruction suivante :

$$\text{begin } S_1; \ldots; S_p \text{ end}$$

3. une instruction "while", une boucle qui répète une instruction S donnée

$$\text{while } R_i \neq 0 \text{ do } S$$

l'instruction S est exécutée tant que le contenu du registre numéro i n'est pas nul, s'il est nul on passe à l'instruction suivante.

4. une instruction "for", une boucle de répétition bornée d'une instruction S donnée

$$\text{for } i = 1 \text{ to } R_j \text{ do } S$$

L'instruction S est exécutée un nombre de fois égal à l'entier contenu dans le registre R_j *avant exécution de ces instructions*, puis l'on passe à l'instruction suivante. Même si le registre R_j est modifié par l'instruction S, le nombre de répétitions de l'instruction n'est pas modifié.

On appelle programme while les programmes structurés qui n'utilisent pas l'instruction for.

L'exécution d'une instruction est plus complexe que pour un programme goto : chaque instruction peut avoir en fait la même complexité qu'un programme. En particulier on peut avoir imbrication des while et des for. L'instruction halt est devenue inutile : les instructions du programme sont exécutées l'une après l'autre, mais la machine ne termine pas toujours son calcul pour autant, puisqu'une boucle while peut ne pas terminer. C'est d'ailleurs la seule instruction susceptible de conduire le programme à ne pas terminer.

Les fonctions partielles calculables sur machines à registres avec programme structuré ou programme while se définissent de la même façon qu'au paragraphe précédent, on suppose de plus que la machine a toujours au moins autant de registres que n l'arité de la fonction.

Lemme 1.3.5 *Si une fonction de* $\mathbb{N}^n \to \mathbb{N}$ *est calculée par une machine avec programme structuré, elle est calculable par une machine avec programme* while.

Démonstration. On suppose $n \leq k$. On procède par induction sur la définition des instructions (les boucles for peuvent être imbriquées). On montre le résultat en simulant chaque instruction d'un programme structuré sur une machine M à k registres par une instruction ou une suite d'instructions sur une machine à $k+s$ registres, où s est le nombre d'instructions for dans le programme. À chaque instruction for est associé de façon univoque un registre R_{k+f}, $1 \leq f \leq s$. On suppose que S est simulée par une séquence d'instructions sans for $\mathscr{S}$, et on simule l'instruction

$$\text{for } i = 1 \text{ to } R_j \text{ do } S$$

par la séquence :

$$R_{k+f} := R_j$$
$$\text{while } R_{k+f} \neq 0 \text{ begin } R_{k+f} := R_{k+f} - 1; \mathscr{S} \text{ end}$$

On doit bien recopier le registre R_j pour le laisser dans le même état à la fin du calcul. D'autre part la suite d'instructions $\mathscr{S}$ doit laisser le registre R_{k+f} dans le même état. ■

Proposition 1.3.6 *Si une fonction de* $\mathbb{N}^n \to \mathbb{N}$ *est calculée par une machine à registres avec programme structuré, elle est calculable par une machine à registres avec programme* goto.

Démonstration. D'après le lemme il suffit de le montrer pour les programmes while. On le montre par induction. Il suffit de simuler chaque instruction d'un programme while par une séquence d'instructions avec goto.

1. Les instructions d'assignations sont des instructions de base des machines à registres ou sont simulables d'après la proposition 1.3.3.

2. On suppose que les instructions $S_1,\dots,S_p$ sont simulées par les suites d'instructions $\mathscr{S}_1,\dots,\mathscr{S}_p$, appelons $\mathscr{S}$ la suite obtenue en les concaténant. Alors la séquence

$$\mathsf{begin}\ S_1;\dots;S_p\ \mathsf{end}$$

est simulée par la suite d'instructions $\mathscr{S}$.

3. On suppose que l'instruction S est simulée par la suite d'instructions $\mathscr{S}$ de longueur s. Alors l'instruction

$$\mathsf{while}\ R_i \neq 0\ \mathsf{do}\ S$$

est simulée par la séquence (les lignes sont numérotées pour que ce soit plus clair) :

$$\begin{array}{rl}
\vdots & \vdots \\
l & \mathsf{if}\ R_i = 0\ \mathsf{goto}\ l+(s+2) \\
\vdots & \mathscr{S} \\
l+s+1 & \mathsf{goto}\ l \\
l+s+2 & \dots \\
\vdots & \vdots
\end{array}$$

Via la simulation, les programmes structurés apparaissent comme des cas particuliers de programme goto, au sens où ils restreignent l'usage de l'instruction goto. En fait les machines avec programmes goto et programmes structurés calculent la même classe de fonctions.

Cette équivalence apparaitra aussi comme conséquence d'un résultat ultérieur : on montrera que cette classe des fonctions est celle de toutes les fonctions μ-récursives partielles.

Pour la preuve directe, on utilisera le lemme suivant qui étend le jeu d'instructions utilisable par un programme structuré.

Lemme 1.3.7 *Soit une fonction partielle de $\mathbb{N}^n \to \mathbb{N}$ calculable par une machine à registres avec programme structuré utilisant en plus l'instruction*

$$\mathsf{if}\ R_i = 0\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2$$

qui exécute l'instruction S_1 ou S_2 suivant le résultat du test à zéro de R_i. Alors f est calculable par une machine à registres avec programme structuré.

Démonstration. On montre le résultat par induction sur le nombre d'instructions if dans le programme. Pour chaque instruction if, on a besoin de deux nouveaux registres que l'on ajoute à la machine.

Soit M une machine calculant f et utilisant if. On chosit une instruction :

$$\mathsf{if}\ R_i = 0\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2$$

telle que S_1 et S_2 ne contiennent pas de if. soient C_1 et C_2 les deux nouveaux registres associés. Alors cette instruction est simplement remplacée par

$$\begin{array}{l}
\mathsf{begin} \\
\quad C_1 := 1; \\
\quad C_2 := 1; \\
\quad \mathsf{for}\ j = 1\ \mathsf{to}\ R_i\ \mathsf{do}\ C_1 := 0; \\
\quad \mathsf{for}\ j = 1\ \mathsf{to}\ C_1\ \mathsf{do}\ \mathsf{begin}\ \mathscr{S}_1; C_2 := 0\ \mathsf{end}; \\
\quad \mathsf{for}\ j = 1\ \mathsf{to}\ C_2\ \mathsf{do}\ \mathscr{S}_2; \\
\mathsf{end}
\end{array}$$

■

Proposition 1.3.8 *Si une fonction de $\mathbb{N}^n \to \mathbb{N}$ est calculable par une machine à registres avec programme* goto, *elle est calculable par machine à registres avec programme structuré.*

On obtient ce résultat comme conséquence de ceux obtenus par codage à la section 1.5 (en particulier la proposition 1.5.5). Cependant la démonstration directe qui suit est informative en soi.

Démonstration. Soit $f : \mathbb{N}^n \to \mathbb{N}$ une fonction calculable par une machine à registre avec programme goto M dont la séquence d'instructions est $S_1, ..., S_p$. On suppose que M utilise au plus k registres avec $k \geq n$. On suppose sans perte de généralité que S_p est l'instruction halt. On va construire un programme structuré N à $k+p$ registres calculant la fonction f. On appelera $I_1, ..., I_p$ les p registres supplémentaires. La signification de $I_i = 0$ est que l'instruction courante du programme est le numéro i. Un seul registre I_i, $i \leq p$, peut être à 0 à un instant donné du temps. Tous les autres doivent avoir la valeur 1.

$I_1 := 0; I_2 := 1; ...; I_p := 1$
while $I_p \neq 0$ do
  if $I_1 = 0$ then $\tilde{S}_1$
  else if $I_2 = 0$ then $\tilde{S}_2$
  $\vdots$
  else if $I_{p-1} = 0$ then $\tilde{S}_{p-1}$

où chaque instruction $\tilde{S}_i$ se définit à partir de S_i comme l'indique le tableau suivant.

S_i	$\tilde{S}_i$
$R_j := R_j + 1$	begin $R_j := R_j + 1; I_{i+1} := 0; I_i := 1$ end
$R_j := R_j - 1$	begin $R_j := R_j - 1; I_{i+1} := 0; I_i := 1$ end

if $R_j = 0$ goto p	begin
	  if $R_j = 0$ then $I_p := 0$ else $I_{i+1} := 0$;
	  $I_i := 1$
	end

■

Le résultat précédent présente une sorte de forme normale de programme structuré (« une boucle suffit »). Toutefois, le programme structuré fourni par la preuve est construit en fonction du programme simulé, et dépend donc de celui-ci. On verra un résultat plus fort : on peut obtenir un programme structuré de la même forme qui est « universel » pour toutes les machines, la machine étant passée en argument via codage.

Le lemme suivant sera commode pour montrer que les fonctions μ-récursives sont calculable par machine.

Définition 1.3.9 Une machine est dite *propre* quand elle termine son calcul avec tous ses registres à l'exception du registre de sortie R_0 dans le même état qu'au début du calcul.

Lemme 1.3.10 *Si une fonction est calculable par machine à registres avec programme structuré, alors elle est calculable par une machine propre (avec programme structuré).*

Démonstration. On suppose, en ajoutant éventuellement des registres, que le nombre initial de ceux-ci est supérieur ou égal au nombre d'arguments de la fonction calculée. Il suffit de copier les registres (sauf celui de sortie), de calculer sur ces copies sans toucher aux registres initiaux, puis de remettre les copies à 0. Soit M une machine à $k+1$ registres $R_0, \ldots, R_k$, dont le programme est une suite d'instructions $(S_1, \ldots, S_s)$ qui calcule f. On construit une machine propre à $2k+1$ registres $R_0, \ldots, R_k, R'_1, \ldots, R'_k$ (R'_i pour R_{i+k}) de la façon suivante :

$R'_1 := R_1$
$\vdots$
$R'_k := R_k$
$S_1[R'_i/R_i]$
$\vdots$
$S_s[R'_s/R_s]$
$R'_1 := 0$
$\vdots$
$R'_k = 0$

■

On ne se servira pas de ce lemme pour les machines avec programme goto, mais il reste valide : il faudrait alors décaler les goto dans les instructions supplémentaires.

Exercice 11

1. Simuler directement l'instruction for par un programme goto.
2. Simuler directement l'instruction

   repeat S until $R_i = 0$

   par un programme goto.

1.4 Les fonctions μ-récursives sont calculables par machines

On montre maintenant comment calculer n'importe quelle fonction partielle μ-récursive par un programme structuré.

1.4.1 Les fonctions partielles μ-récursives

Proposition 1.4.1 *Les fonction partielles μ-récursives sont calculables par machine à registres avec programme structuré. Par conséquent elles sont calculables par machine à registres avec programme* while *et programme* goto.

Démonstration. On procède par induction sur la définition des fonctions partielles μ-récursives. On a déjà traité en exemple les fonctions de base, la fonction nulle $\lambda x.0$, les projections p_k^i et la fonction successeur. On vérifie que les instruction utilisées sont bien celles des programmes structurés.

composition Supposons que les fonctions partielles $g_1,\dots,g_k : \mathbb{N}^n \to \mathbb{N}$ et $h : \mathbb{N}^k \to \mathbb{N}$ sont calculables par des machines $M_1,\dots,M_k$ et M_0, dont les programmes sont les suites d'instructions $\mathscr{S}_1,\dots,\mathscr{S}_k$ et $\mathscr{S}_0$. On suppose ces machines propres d'après le lemme 1.3.10. On suppose que chacune de ces machines utilisent au plus $m+1$ registres, et que $m \geq n$ et $m \geq k$. On construit une machine M qui utilise $m+k+1$ registres que l'on va noter $R_0, R_1, \dots, R_m, H_1, \dots, H_k$. Elle va calculer successivement $g_1(x_1,\dots,x_n)$,...,$g_k(x_1,\dots,x_n)$ dont elle va stocker les valeurs dans les registres $H_1,\dots,H_k$, puis calculer $f(H_1,\dots,H_k)$. Voici le programme de cette machine

$$
\begin{array}{l}
\mathscr{S}_1 \\
H_1 := R_0 \\
R_0 := 0 \\
\vdots \\
\mathscr{S}_k \\
H_k := R_0 \\
R_0 := 0 \\
R_1 := H_1 \\
\vdots \\
R_k := H_k \\
R_{k+1} := 0 \\
\vdots \\
R_m := 0 \\
\mathscr{S}_0
\end{array}
$$

Remarquez que dès que l'une des machines $M_1,\dots,M_k$ ne termine pas la machine M ne termine pas.

Récurrence primitive Soit M_1 une machine propre à k_1+1 registres dont le programme est $\mathscr{S}_1$ et qui calcule la fonction partielle $g : \mathbb{N}^n \to \mathbb{N}$ $(n \leq k_1)$. Soit M_2 une machine propre à k_2+1 registres dont le programme est $\mathscr{S}_2$ et qui calcule la fonction partielle $h : \mathbb{N}^{n+2} \to \mathbb{N}$ $(n+2 \leq k_2)$. Soit un

entier m tel que $m \geq k_1$ et $m \geq k_2$. La fonction f est définie par récurrence primitive à partir de g et h :

$$\begin{aligned} f(\overline{a},0) &= g(\overline{a}) \\ f(\overline{a},x+1) &= h(\overline{a},x,f(\overline{a},x)). \end{aligned}$$

La fonction f est calculée par une machine à $m+2$ registres dont voici les instructions :

$R_{m+1} := R_{n+1}$
$R_{n+1} := 0$
$\mathscr{S}_1$
for $i = 1$ to R_{m+1} do begin $R_{n+2} := R_0$; $R_0 := 0$; $\mathscr{S}_2$; $R_{n+1} := R_{n+1} + 1$ end

On calcule donc successivement $f(\overline{a},0)$ (avant la boucle for) puis dans l'ordre $f(\overline{a},1),\ldots,f(\overline{a},n)$. Dès que l'une de ces valeurs n'est pas définie, la machine ne termine pas, ce qui est bien le comportement souhaité.

Minimisation Soit M_0 une machine propre à $k+1$ registres dont le programme est $\mathscr{S}_0$ qui calcule la fonction partielle $g : \mathbb{N}^{n+1} \to \mathbb{N}$, avec $n+1 \leq k$. La fonction f :

$$f(x_1,\ldots,x_n) = \mu t.[g(x_1,\ldots,x_n,t) = 0]$$

est calculée par la machine à $k+1$ registres suivante :

$R_0 := R_0 + 1$
while $R_0 \neq 0$ do begin $R_0 := 0$; $\mathscr{S}_0$; $R_{n+1} := R_{n+1} + 1$ end
$R_{n+1} := R_{n+1} - 1$
$R_0 := R_{n+1}$ ■

1.4.2 Les fonctions récursives primitives

Si on examine la preuve, on se rend facilement compte que le while n'intervient que pour la minimisation. Le for suffit pour la récurrence primitive. Appelons programme for les programmes structurés qui n'utilisent pas de while, on a donc :

Proposition. Les fonctions récursives primitives sont calculables par programme for.

La réciproque de ce résultat est vraie, c'est-à-dire que :

Proposition 1.4.2 *Une fonction est récursive primitive si et seulement si elle est calculable par programme* for.

Il faut bien remarquer que les boucles for que nous avons définies ne peuvent modifier la variable d'itération lors du calcul, alors que c'est possible par exemple dans le langage C. Sans cette condition le résultat est manifestement faux (le programme ne termine pas forcément).

Exercice 12 Terminer la démonstration de la proposition précédente, soit si une fonction est calculable par programme for, elle est récursive primitive. Procéder par induction sur le niveau d'imbrication des boucles for. Montrer que le contenu de chaque registre de la machine est une fonction récursive primitive des entrées en utilisant le schéma de récurrences mutuelles (exercice 5 page 10).

1.5 Les fonctions calculables par machine sont μ-récursives

Nous allons maintenant montrer que les fonctions partielles calculables par machine goto sont μ-récursives. La méthode consiste à coder par des entiers les configurations d'une machine quelconque (l'état de la machine à un instant donné), et à refléter dans l'arithmétique par des fonctions μ-récursives le fonctionnement de cette machine. On parle parfois d'*arithmétisation*. Le premier exemple d'arithmétisation est celui des formules de l'arithmétique par Gödel dans son article paru en 1931.

Là on va arithmétiser plus que nécessaire pour le résultat annoncé : en codant également les machines par des entiers, plutôt que de montrer l'existence d'une fonction μ-récursive ad hoc décrivant le

fonctionnement de chaque machine, on montre que ce fonctionnement se décrit de façon uniforme en prenant le code de la machine en argument. La démonstration n'est pas plus compliquée, et cette arithmétisation plus poussée conduit aux résultats du chapitre suivant, en particulier à l'existence d'une machine universelle, une machine qui peut simuler n'importe quelle machine (disons en fixant le nombre d'entrées) en lui fournissant le bon argument.

1.5.1 Machine, état d'une machine

Une machine est déterminée par :

1. son nombre de registres ;
2. son programme qui est une suite finie d'instructions.

L'état d'une machine est déterminée par

1. un index de lecture (entier) : le numéro de l'instruction à exécuter ;
2. la suite finie des contenus des registres.

Le code d'une machine ou d'un état de machine est un entier, la fonction de codage doit être injective, mais pas nécessairement surjective.

Préliminaires et notations sur les fonctions récursives primitives

On va utiliser les différents codages récursifs primitifs des couples, k-uplets et listes (suite finies) définis à la section 1.1.4 page 7, ainsi que des fonctions usuelles sur ceux-ci. Les arguments de la fonction $\mathrm{nth}(l,i)$ qui calcule le i-ème élément de la suite de code l sont notés dans cet ordre.

On utilisera maintenant systématiquement $\langle , \rangle$ pour les bijections de Cantor α_k :

$$\alpha_k(x_1,\dots,x_k) = \langle x_1,\dots,x_k\rangle \ .$$

Par définition des α_i :

$$\langle x_1, \langle x_2,\dots,x_k\rangle\rangle = \langle x_1,\dots,x_k\rangle$$

et les projections vérifient $\pi_1 = \pi_1^2 = \pi_1^3 = \dots$ et plus généralement :

$$\pi_i^n = \pi_i^m \text{ pour } i < n \leq m$$

Comme α_k est une bijection on obtient une définition correcte de fonction en écrivant :

$$f(x_1,\dots,x_{i-1},\langle y_1,\dots,y_k\rangle,x_{i+1},\dots,x_n) = h(x_1,\dots,x_{i_1},y_1,\dots,y_k,x_{i+1},\dots,x_n)$$

et si h est récursive primitive alors f est récursive primitive. En effet cette définition équivaut à :

$$f(x_1,\dots,x_n) = h(x_1,\dots,x_{i-1},\pi_1^k(x_i),\dots,\pi_k^k(x_i),x_{i+1},\dots,x_n) \ .$$

Codage des instructions

On a quatre sortes d'instructions, chacune pouvant être paramétrée par un entier (le numéro de registre pour l'incrémentation et la décrémentation), ou un couple d'entiers (le numéro de registre et le numéro de ligne pour le goto conditionnel). On va donc coder chaque instruction par le code d'un couple, on note $\ulcorner S \urcorner$ l'entier qui code l'instruction S :

$$\begin{aligned}
\ulcorner R_i := R_i + 1 \urcorner &= \langle 0, i\rangle \\
\ulcorner R_i := R_i - 1 \urcorner &= \langle 1, i\rangle \\
\ulcorner \text{if } R_i = 0 \text{ goto } n \urcorner &= \langle 2, \langle i, n\rangle\rangle (= \langle 2, i, n\rangle) \\
\ulcorner \mathsf{halt} \urcorner &= \langle 3, 0\rangle
\end{aligned}$$

De fait, le codage n'a pas besoin d'être fonctionnel, et nous considérerons parfois dans la suite que tout entier c tel que $\pi_1^2(c) \geq 3$ est le code d'une instruction halt.

Codage des machines

Le code m d'une machine à $k+1$ registres de programme $S_0,\ldots,S_s$ est l'entier :

$$m = \langle k, [\ulcorner S_1 \urcorner;\ldots;\ulcorner S_s \urcorner]\rangle.$$

Une machine possède au moins un registre, la première projection est donc le nombre de registres moins 1. Le codage est évidemment injectif, mais pas surjectif, ce qui n'est pas gênant. On n'aura même pas besoin du résultat de l'exercice suivant.

Exercice 13 Montrer que l'ensemble des entiers qui sont des codes de machine est récursif primitif.

Codage de l'état d'une machine

On code l'état d'une machine dont les contenus des registres sont dans l'ordre $R_0, R_1, \ldots, R_k$ et l'index de lecture est i par l'entier :

$$e = \langle i, [R_0; R_1; \ldots; R_k]\rangle$$

1.5.2 Le calcul

Supposons que la machine de code m prenne en entrée n entiers. Pour coder le calcul, il nous faut essentiellement les deux fonctions et le prédicat suivant :

— Une fonction d'initialisation $\text{init}_n : \mathbb{N}^{n+1} \to \mathbb{N}$. Cette fonction calcule en fonction du code m de la machine et des entrées $x_1,\ldots,x_n$ l'état initial de cette machine (au départ du calcul).
— Une fonction de transition $\text{tr} : \mathbb{N}^2 \to \mathbb{N}$. Cette fonction calcule en fonction du code m d'une machine et de l'état de code e de cette machine l'état de la machine après une étape de calcul à partir de l'état de code e.
— Le prédicat Halt de terminaison à deux arguments est vrai pour les couples (m,e) où l'état codé par e indique que la machine codée par m est en fin de calcul.

Remarquez qu'il ne s'agit pas vraiment de définitions : on ne demande rien si les entrées ne sont pas cohérentes, ce qui est le cas par exemple pour ces fonctions et ce prédicat quand la première entrée n'est pas le code d'une machine, ou encore pour la fonction de transition et le prédicat de terminaison quand la seconde entrée n'est pas le code d'un état possible de la machine codée par la première entrée.

Il est intuitivement clair qu'on peut calculer de façon mécanique deux fonctions et un prédicat convenables. On montre explicitement dans les sections qui suivent qu'on peut les définir de façon récursive primitive.

Fonction d'initialisation

On définit d'abord une fonction auxiliaire qui crée en fonction de k la liste des $k+1$ premiers éléments de la suite infinie $0, x_1, \ldots, x_n, 0, \ldots$. Ensuite init_n construit l'état initial de la machine de code m, à savoir un index de lecture à 0, et si celle-ci possède $k+1$ registres ($k = \pi_1^2(m)$), le code de la liste de longueur $k+1$ $[0; x_1; \ldots; x_n; 0; \ldots; 0]$ (liste éventuellement tronquée si $k < n$) :

$$\begin{aligned}
&aux(x_1,\ldots,x_n,0) = [0] = 0 :: []\\
&aux(x_1,\ldots,x_n,1) = [0;x_1] = 0 :: x_1 :: []\\
&\vdots\\
&aux(x_1,\ldots,x_n,n) = [0;x_1;\ldots;x_n] = 0 :: x_1 :: \ldots :: x_n :: []\\
&aux(x_1,\ldots,x_n,n+r+1) = aux(x_1,\ldots,x_n,n+r)@[0]\\
&\text{init}_n(\langle k,s\rangle, x_1,\ldots,x_n) = \langle 0, aux(x_1,\ldots,x_n,k)\rangle
\end{aligned}$$

Lemme 1.5.1 *la fonction* $\text{init}_n(m,x_1,\ldots,x_n)$ *définie ci-dessus est récursive primitive. Elle calcule, quand m est le code d'une machine, l'état initial de la machine de code m avec $x_1,\ldots,x_n$ en entrées.*

Démonstration. La fonction se comporte comme souhaité si m est le code d'une machine. Le schéma de récurrence utilisé est récursif primitif. En effet on le ramène à une récurrence primitive en utilisant dans l'étape de récurrence n fois le «si ... alors ... sinon ...» sur des conditions récursives primitives (singleton). Notez bien que n est une constante. ■

Fonction de transition

On va avoir besoin pour écrire la fonction de transition de modifier le registre numéro i, ce qui demande, pour le codage considéré, de modifier le i-ème élément d'une liste.

Proposition 1.5.2 *Les fonctions* inc, *et* dec *qui, appliquées à un entier i et au code d'une liste non vide, incrémente de* 1 *le $i+1$-ème élément de la liste pour* inc, *respectivement décrémente de* 1 *ce $i+1$-ème élément pour* dec, *et qui ne changent pas leur argument sinon, sont récursives primitives.*

Démonstration. On peut utiliser le schéma de récurrence avec substitution de paramètre (voir exercice 7 page 11), on a :

$$\begin{array}{ll} \text{si } \mathrm{len}(l) \leq n & \text{alors } \mathrm{inc}(n,l) = l \\ & \text{sinon} \\ & \quad \mathrm{inc}(0,l) = (\mathrm{hd}(l)+1) :: \mathrm{tl}(l) \\ & \quad \mathrm{inc}(n+1,l) = \mathrm{hd}(l) :: \mathrm{inc}(n, tl(l)) \end{array}$$

De même pour dec. ■

Par ailleurs on peut calculer de façon récursive primitive l'instruction courante : $\mathrm{inst}(m,e)$ est le code de l'instruction de la machine de code $m = \langle n,s\rangle$ auquel renvoie l'index de lecture indiqué par l'état de code $e = \langle i,r\rangle$ (si l'index de lecture i est supérieur au nombre d'instructions, on considère qu'il renvoie à la dernière instruction) :

$$\mathrm{inst}(\langle n,s\rangle, \langle i,r\rangle) = \mathrm{nth}(s, \mathrm{inf}(i, \mathrm{len}(s))) .$$

On décompose maintenant la fonction de transition en ses deux projections, soient tr_1 et tr_2, et on montre que l'on peut définir chacune de façon récursive primitive.

— $\mathrm{tr}_1(m,e)$ donne le numéro de l'instruction après exécution de l'instruction indiquée par $e = \langle i,r\rangle$:

$$\begin{array}{lll} \text{si } \pi_1^2(\mathrm{inst}(m,\langle i,r\rangle)) \in \{0,1\} & & \text{si incrémentation ou décrémentation} \\ \quad \text{alors } \mathrm{tr}_1(m,\langle i,r\rangle) = i+1 & & \text{on passe à l'instruction suivante} \\ \quad \text{sinon si } \pi_1^2(\mathrm{inst}(m,\langle i,r\rangle)) = 2 & & \text{si goto} \\ \qquad \text{alors si } \mathrm{nth}(r, \pi_2^3(\mathrm{inst}(m,\langle i,r\rangle))) = 0 & & \text{si } R_j = 0 \\ \qquad\quad \text{alors } \mathrm{tr}_1(m,\langle i,r\rangle) = \pi_3^3(\mathrm{inst}(m,\langle i,r\rangle)) & & \text{aller à la ligne indiquée} \\ \qquad\quad \text{sinon } \mathrm{tr}_1(m,\langle i,r\rangle) = i+1 & & \text{sinon on passe à l'instruction suivante} \\ \qquad \text{sinon } \mathrm{tr}_1(m,\langle i,r\rangle) = i & & \text{sinon (halt) arrêt} \end{array}$$

— $\mathrm{tr}_2(m,e)$ donne la liste des contenus des registres après exécution de l'instruction indiquée par $e = \langle i,r\rangle$:

$$\begin{array}{lll} \text{si } \pi_1^2(\mathrm{inst}(m,\langle i,r\rangle)) = 0 & & \text{si incrémentation} \\ \quad \text{alors } \mathrm{tr}_2(m,\langle i,r\rangle) = \mathrm{inc}(\pi_2^2(\mathrm{inst}(m,\langle i,r\rangle)), r) & & R_j := R_j + 1 \\ \quad \text{sinon si } \pi_1^2(\mathrm{inst}(m,\langle i,r\rangle)) = 1 & & \text{si décrémentation} \\ \qquad \text{alors } \mathrm{tr}_2(m,\langle i,r\rangle) = \mathrm{dec}(\pi_2^2(\mathrm{inst}(m,\langle i,r\rangle)), r) & & R_j := R_j - 1 \\ \qquad \text{sinon } \mathrm{tr}_2(m,\langle i,r\rangle) = r & & \text{sinon (goto ou halt) les registres ne sont pas modifiés} \end{array}$$

Les deux fonctions tr_1 et tr_2 sont bien définies de façon récursive primitive, la fonction de transition

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

est donc récursive primitive. Résumons les résultats de cette section.

Lemme 1.5.3 *Les fonctions* inst *et* tr *définies ci-dessus sont primitives récursives. Quand m est le code d'une machine M et e le code d'un état E possible pour M,* $\mathrm{inst}(m,e)$ *est le code de l'instruction courante du programme de M selon l'état E,* $\mathrm{tr}(m,e)$ *est le code de l'état de M obtenu après exécution de cette instruction.*

Prédicat de terminaison

On rappelle que l'état d'une machine est terminal si l'instruction indiquée par l'état est l'instruction halt. On peut donc définir le prédicat de terminaison Halt ainsi :

$$\mathrm{Halt}(m,e) \equiv \pi_1^2(\mathrm{inst}(m,e)) > 2$$

Lemme 1.5.4 *Le prédicat binaire* Halt *défini ci-dessus est récursif primitif. Il est vrai pour les couples* (m,e) *où l'état codé par e indique que la machine codée par m est en fin de calcul, quand m est le code d'une machine, et e le code d'un état pour m.*

Les entiers qui ne sont pas des codes de machines

Nous avons défini ci-dessus les fonctions d'initialisation et de transition et le prédicat de terminaison pour tous les entiers. Parmi ceux-ci, certains ne codent pas de machine. Cela n'a pas d'importance : ces fonctions décrivent de toute façon un comportement mécanique, et finalement nous allons montrer que les fonctions partielles calculables par une classe plus étendue (au moins en apparence) de machines sont μ-récursives.

Nous décrivons maintenant le comportement de ces « machines » étendues, ce qui ne sera vraiment utile qu'au paragraphe 2.2.3.

Pour un entier donné codant une machine, le nombre de registres est bien défini ainsi que le programme comme suite, il se peut simplement que des éléments de la suite ne correspondent pas à des instructions valides.

Nous avons déjà supposé (Paragraphe 1.5.1) que tout entier c tel que $\pi_1^2(c) > 2$ codait une instruction halt, ce qui correspond bien au choix du prédicat de terminaison. La machine peut avoir des instructions halt n'importe où dans le programme, et la dernière instruction n'est pas forcément un halt, ce qui généralise un peu la notion initiale. La machine s'arrête dès qu'elle atteint un halt. Si la dernière instruction est exécutée et que ce n'est pas un halt ou un goto, alors cette instruction est répétée indéfiniment (voir définition de inst) : la machine ne s'arrête pas.

Les instructions d'incrémentation et de décrémentation ($\pi_1^2(c) = 0,1$) pourraient concerner un numéro de registre qui n'apparait pas dans la machine. Dans ce cas on considère que l'instruction ne fait rien en dehors de passer à l'instruction suivante. C'est bien ce que code tr (voir la définition des fonctions inc et dec).

La condition de l'instruction goto peut porter sur un registre qui n'apparait pas dans la machine : on le considère comme un goto inconditionnel (voir définition de tr_1). Elle peut renvoyer à une ligne de programme d'index supérieur à la longueur du programme. Dans ce cas cela revient à renvoyer à la dernière ligne du programme (voir définition de inst).

Temps de calcul

On a tout ce qu'il faut maintenant pour définir en fonction du code de la machine et des entrées le temps de calcul, c'est à dire le nombre d'étapes jusqu'à ce que la machine termine. Il se calcule par minimisation, et peut ne pas être défini, si la machine ne termine pas.

On définit d'abord de façon récursive primitive la fonction $\mathrm{st}_n(m,x_1,\dots,x_n,t)$ qui calcule l'état de la machine de code m au bout de t étapes de calcul avec les entrées $x_1,\dots,x_n$:

$$\begin{aligned}\mathrm{st}_n(m,x_1,\dots,x_n,0) &= \mathrm{init}_n(m,x_1,\dots,x_n)\\ \mathrm{st}_n(m,x_1,\dots,x_n,t+1) &= \mathrm{tr}(m,\mathrm{st}_n(m,x_1,\dots,x_n,t))\end{aligned}$$

Enfin le prédicat de terminaison $H_n(m,x_1,\dots,x_n,t)$, qui indique que la machine de code m avec en entrée $x_1,\dots,x_n$ s'est arrêtée au bout de t étapes de calcul, se définit de façon primitive récursive.

$$H_n(m,x_1,\dots,x_n,t) \equiv \mathrm{Halt}(m,\mathrm{st}_n(m,x_1,\dots,x_n,t))$$

Le *temps de calcul* de la machine de code m pour les entrées $x_1,\dots,x_n$ est donc obtenu par minimisation, c'est :

$$\mu t.H_n(m,x_1,\dots,x_n,t)$$

Étant donné un état e, le contenu du registre R_0 est donné par $\mathrm{hd}(\pi_2^2(e))$. La fonction calculée par la machine m est donc définie par :

$$\mathrm{hd}\circ\pi_2^2\circ\mathrm{st}_n(m,x_1,\dots,x_n,\mu t.H_n(m,x_1,\dots,x_n,t))$$

Elle est partielle μ-récursive. Remarquez bien que l'argument m est entier quelconque, pas forcément un « vrai » code d'une machine. Résumons les résultats obtenus.

Proposition 1.5.5 *Pour tout entier $n \geq 1$, il existe une fonction récursive primitive $U_n : \mathbb{N}^{n+2} \to \mathbb{N}$, et un prédicat récursif primitif H_n à $n+2$ arguments tels que, si m est le code d'une machine M avec programme* goto, *alors :*

- $\mathrm{st}_n(m, x_1, \ldots, x_n, t)$ *est le code de l'état de la machine M avec en entrées $x_1, \ldots, x_n$ au bout de t étapes de calcul ;*
- $H_n(m, x_1, \ldots, x_n, t)$ *ssi M avec en entrées $x_1, \ldots, x_n$ s'arrête au bout de t étapes de calcul.*
- *La fonction partielle n-aire f calculée par M est μ-récursive et vérifie :*

$$f(x_1, \ldots, x_n) = U_n(m, x_1, \ldots, x_n, \mu t.H_n(m, x_1, \ldots, x_n, t))$$

Corollaire 1.5.6 *Toute fonction partielle calculable par machine à registres avec programme* goto, *avec programme structuré, ou avec programme* while *est μ-récursive.*

Démonstration. D'après la proposition précédente et la proposition 1.3.8. ■

Du corollaire précédent et de la proposition 1.4.1 on déduit l'équivalence des tous les modèles de calcul introduits jusqu'ici pour les fonctions partielles.

Théorème 1.5.7 *Les propositions suivantes sont équivalentes :*

- *la fonction partielle f est μ-récursive ;*
- *la fonction partielle f est calculable par machine à registres avec programme* goto *;*
- *la fonction partielle f est calculable par machine à registres avec programme* while *;*
- *la fonction partielle f est calculable par machine à registres avec programme structuré.*

Temps de calcul d'une fonction récursive primitive

Les fonctions récursives primitives définissent d'une certain façon une classe de complexité (très étendue !) comme le montrent les propositions qui suivent.

Proposition 1.5.8 *Si le temps de calcul (sur une machine à registres) d'une machine est borné par une fonction récursive primitive (en fonction de ses entrées) alors la fonction calculée par la machine est primitive récursive.*

Démonstration. On reprend la Proposition 1.5.5. On peut borner la minimisation par une fonction récursive primitive, le prédicat H_n et la fonction U_n sont récursifs primitifs. ■

La réciproque de cette proposition est intuitivement vraie, mais plus pénible à démontrer.

Proposition 1.5.9 *Toute fonction récursive primitive est calculable par une machine dont le temps de calcul est une fonction récursive primitive des entrées.*

Démonstration. (indications). On montre d'abord que le temps de calcul d'une machine avec programme for est une fonction primitive récursive des entrées : c'est essentiellement le nombre de pas dans les boucles for, une succession de boucles for correspond à une addition, une imbrication à une multiplication. Le nombre de pas d'une boucle est le contenu d'un registre qui est une fonction primitive récursive des entrées (voir exercice 12). On montre ensuite que le temps de calcul reste une fonction récursive primitive par traduction d'un programme for en programme goto. ■

Évidemment une machine peut calculer une fonction récursive primitive sans que son temps de calcul soit une fonction récursive primitive des entrées ! Par exemple la fonction nulle est calculée par une machine obtenue en ajoutant en fin de programme l'instruction $R_0 := 0$ pour n'importe quelle machine calculant une fonction totale.

1.5.3 Thèse de Church

Nous venons de voir que les différentes classes de machines à registres et les fonctions μ-recursives définissent la même notion de fonction calculable. La méthode utilisée pour montrer que les fonctions partielles calculables par machine sont μ-récursives est tout à fait générale. Dès que l'on a une notion de machine (ou de programme) avec un calcul qui se ramène à une succession d'états, et un ou des états terminaux, on a juste besoin :

— de coder les machines et les états des machines ;
— de montrer que les fonctions d'initialisation et de transition induisent sur les codes des fonctions totales qui sont μ-récursives, et que le prédicat d'arrêt est décidable (au sens des fonctions μ-récursives).

Bien d'autres modélisations de la notion de fonction calculable existent. On citera entre autres les machines de Turing, le lambda calcul, des systèmes basés sur des principes de substitution de mots (Post), des systèmes équationnels (Herbrand-Gödel), des machines à pointeurs (Kolmogorov-Uspenski, Schönhage) etc. Tous ces modèles sont équivalents du point de vue de la calculabilité. À chaque fois la méthode résumée ci-dessus peut s'utiliser pour montrer que les fonctions partielles calculables avec l'un de ces modèles de calcul sont μ-récursives[5]. On a même du mal à imaginer comment une modélisation avec calcul explicite pourrait y échapper, même si les détails techniques peuvent s'avérer fastidieux. Il faut vérifier bien entendu à chaque fois la réciproque, mais si celle-ci n'est pas vérifiée on a simplement un modèle de calcul moins puissant.

En résumé cette série de résultats, mais aussi la grande généralité de la méthode mise en œuvre, semblent confirmer la conjecture suivante, dite thèse de Church.

Thèse de Church *Toute fonction calculable est μ-récursive (ou calculable par machine à registres, ou calculable par machine de Turing, etc.)*

Cette affirmation n'est pas démontrable puisqu'elle elle met en relation une notion intuitive (le fait d'être calculable) avec une notion mathématiquement précise (être calculable dans un modèle particulier). Mais on a tout de même donné des arguments pour sa validité, et elle n'a jamais pu être infirmée.

5. D'autres démonstrations par simulations directes des modèles de calcul entre eux sont aussi possibles.