Calculabilité et incomplétude - Notes de cours

Arnaud Durand, Paul Rozière
Paris7 – M2 LMFI

13 novembre 2013
(version provisoire — 14: 50)

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 introduit trois modèles de calcul différents, à savoir :
- les fonctions récursives,
- les machines de Turing,
- les machines RAM (à nombre fini de registres).

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.

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.

La première approche présentée est celle des fonctions 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 est élégante, et elle fournit un intermédiaire commode pour montrer que les diverses notions de fonctions calculables sont équivalentes. Mais la formalisation du calcul (de n'importe quel modèle de calcul suffisamment riche en fait) s'avère très vite indispensable pour prolonger l'étude.

On commence par étudier une première notion plus restreinte, les fonctions récursives primitives.

1.1 Fonctions récursives primitives

On pose $\mathscr{F} = \bigcup_{p\in\mathbb{N}^*} \mathbb{N}^{\mathbb{N}^p}$.

Définition 1.1.1 L'ensemble des *fonctions récursives primitives* est le plus petit sous-ensemble de $\mathscr{F}$ qui satistfait les conditions suivantes :

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

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

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

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 cartéristique d'un prédicat est celle de l'ensemble des uples pour lesquels le prédicat est vrai.

On dira d'un sous-ensemble de $\mathscr{F}$ 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 est clos par opérations récursives primitives, et, dans le cas de l'ensemble des fonctions récursives, ce sera une conséquence de la définition.

1.1.1 Exemples de fonctions récursives primitives

Fonctions constantes Pour $n \in \mathbb{N}$, on note s^n la composition n fois de la fonction successeur. La fonction constante $c_n : \mathbb{N} \to \mathbb{N}$ telle que $c_n(x) = n$ se définit par $c_n(x) = \mathrm{s}^n(\odot(x))$, en utilisant 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_1^3, 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 :

$$\times(x,0) = 0 \text{ et } \times(x,y+1) = +(p_1^3(x,y,+(x,y)), p_3^3(x,y,\times(x,y)))$$

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 **iii₀**. 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 est ne peut être levée en toute généralité : on verra qu'il existe des fonctions comme la fonction d'Ackermann (voir 1.2.2 page 12) 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 (qui plus est « simple ») 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.

Somme et produits limités 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 primitif récursif. 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 Si f et g sont deux fonctions récursives primitives et A est un prédicat récursif primitif alors la fonction h suivante est récursive primitive :

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

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 prédicats récursifs primitifs deux à deux disjoints et $f_1, ..., f_n$ des fonctions récursives primitives alors la fonction h suivante est récursive primitive :

$$\begin{aligned} h(\overline{x}) &= f_1(\overline{x}) \text{ si } \overline{x} \in A_1 \\ h(\overline{x}) &= f_2(\overline{x}) \text{ si } \overline{x} \in A_2 \\ \vdots\ &\ \vdots \\ h(\overline{x}) &= f_n(\overline{x}) \text{ si } \overline{x} \in A_n. \end{aligned}$$

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 :

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

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 :

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

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 :

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

En effet

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

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

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

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.

Plus généralement les propriétés de clôture de ce paragraphe sont valides pour tout sous-ensemble de $\mathscr{F}$ (fonctions à plusieurs arguments entiers partout définies) clos par opérations récursives primitives.

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 opération 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 :

$$p \text{ est premier } \equiv \forall x \leq p\, \forall y \leq p\, \left(x \cdot y = p \rightarrow \left(x = 1 \vee x = p\right)\right) .$$

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 $p : \mathbb{N} \rightarrow \mathbb{N}$ 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 faut 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 fonctions récursives. Il est bien évident que la notion de calcul dépasse de loin ce cadre restreint. Dans un sens, un modèle de calcul général doit pouvoir considérer en entrée des (représentations finies) d'objets de nature très différentes : nombres autres qu'entiers, mots sur un alphabet fini, structures algébriques, graphes, hypergraphes, arbres, fonctions et relations... La notion de calcul a intuitivement un sens dans tous ces contextes. De même, clairement, les fonctions récursives (ou programme) doivent pouvoir, à travers une représentation, être considérées comme des paramètres valides d'autres fonctions récursives (ou programme).

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. Les méthodes utilisées se généralisent à des structures plus complexes, comme on le verra dans la partie 6.2

La bijection de Cantor

On appelle α_2 la bijection de Cantor $\mathbb{N}\times\mathbb{N}$ dans $\mathbb{N}$, définie par :

$$\alpha_2(n,p) = (\sum_{i=0}^{n+p} i) + p = \frac{(n+p+1)(n+p)}{2} + p$$

On vérifie facilement que α_2 est bijective, strictement croissante pour ses deux entrées n et p et qu'elle est récursive primitive [1]. 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) :

$$\alpha_2(\pi_1(c),\pi_2(c)) = c, \quad \pi_1(\alpha_2(n,p)) = n, \quad \pi_2(\alpha_2(n,p)) = p$$

qui sont également récursives primitives. En effet :

$$\pi_1(x) = \mu z \le x.[\exists t \le x\ \alpha_2(z,t) = x] \quad \text{et} \quad \pi_2(x) = \mu t \le x.[\exists z \le x\ \alpha_2(z,t) = x].$$

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

$$\alpha_1(n) = n \quad \text{et} \quad \alpha_{k+1}(n_1,\ldots,n_{k+1}) = \alpha_2(n_1,\alpha_k(n_2,\ldots,n_{k+1}))\,.$$

À nouveau on peut démontrer 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 $\alpha_k(x_1,\ldots,x_k) = \langle x_1,\ldots,x_k\rangle$ 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.

Un codage bijectif des suites

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

$$x :: y = 1 + \alpha_2(x,y)$$

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 inductivement (on note $[a_0;\ldots;a_n] = \mathrm{liste}((a_0,\ldots,a_n))$) :

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

1. De plus, même si ce n'est pas très utile dans un contexte où on ne s'intéresse pas à la complexité, on peut remarquer que, si les entiers sont en base 2, la taille d'un entier représentant un couple est proportionnelle à la somme des tailles de ses composantes

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 :

$$\mathrm{hd}(c) = \mu x < c.\left[\exists y < c\left(x :: y = c\right)\right] \text{ et } \mathrm{tl}(c) = \mu y < c.\left[\exists x < c\left(x :: y = c\right)\right]$$

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. 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 :*

$$f(a_1,\dots,a_p,0) = g(a_1,\dots,a_p)$$
$$f(a_1,\dots,a_p,x+1) = h\left(a_1,\dots,a_p,x,\left[f(a_1,\dots,a_p,x);\dots;f(a_1,\dots,a_p,0)\right]\right).$$

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

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

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

À partir du lemme précédent, on peut montrer que la fonction nthl qui à i et l associe la suite codée par l à partie du $i+1$-$i$ème élément (0 sinon), et la fonction nth qui à i et l associe le $i$ème élément de la suite codée par l (0 sinon), sont récursives primitives

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 $l = [a_0;\dots;a_i;\dots;a_n]$ associe $\mathrm{inc}(i,l) = [a_0;\dots;a_i+1;\dots;a_n]$ quand $i \leq n$, $\mathrm{inc}(i,l) = l$ 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 $\mathrm{p} : \mathbb{N} \to \mathbb{N}$ 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 seq : $\mathscr{S} \to \mathbb{N}$ associe à chaque suite de longueur finie $(x_1,\dots,x_k)$ la valeur suivante

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

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 $\mathrm{seq}(x_1,...,x_k)$ et $\mathrm{seq}(y_1,...,y_h)$ codant des suites renvoie le nombre représentant la concaténation des deux listes $\mathrm{seq}(x_1,...,x_k,y_1,...,y_h)$.

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

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

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

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

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 $a_1,\dots,a_p,l$ codant $(u_i)_{i\le n}$ associe $\mathrm{map}_f(l)$ codant $\big(f(a_1,\dots,a_p,u_i)\big)_{i\le n}$ est récursive primitive.

Montrer que la fonction $concat$ qui à un entier l codant une liste de liste $\big((u_i)_{i\le n_i}\big)_{i\le p}$ associe l'entier $concat(l)$ codant la suite des entiers de chaque suite $(u_i)_{i\le n_i}$ 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 $f:\mathbb{N}^2\to\mathbb{N}$.

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

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

$$\begin{array}{l} F(p,a,0)=g\big(\gamma^p(a)\big) \\ F(p,a,x+1)=h\big(\gamma^{p\dot{-}(x+1)}(a),x,F(p,a,x)\big)\,. \end{array}$$

2. Montrer que :

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

et en déduire que f est récursive primitive.

3. Application : montrer que la fonction inc : $\mathbb{N}^2 \rightarrow \mathbb{N}$ 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 suivante, 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 :

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

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 :

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

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 :

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

1.2 Les fonctions récursives

1.2.1 Évaluation des fonctions récursives primitives

Les fonctions récursives primitives ne résument pas, à elles seules, l'ensemble des fonctions calculables au sens intuitif (dans la suite de ce paragraphe, « calculable » signifie « calculable 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 6.2). 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, telle que, si i est le code d'une fonction récursive primitive à un argument f, 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 $\odot$ et s sont des termes récursifs primitifs d'arité 1 ; pour $k \in \mathbb{N}^*$, $1 \leq i \leq k$, p_i^k est un terme récursif primitif d'arité k ;

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

iii. si g et h sont des termes récursifs primitifs, g d'arité p et h d'arité $p+2$, alors $\mathsf{rec}(g,h)$ 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 $\mathsf{Eval} = \mathsf{Eval}_1$).

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

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

Supposons que les définitions soient codées par des entiers. On suppose de plus qu'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 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.

– 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 à la fonction 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 au plus n occurrences du schéma de récurrence (majoration non optimale, voir plus loin). 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*[2]. Elle se définit par récurrence double avec imbri-

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

cation des appels récursifs :

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

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

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

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

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

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.

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

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 $\text{Ack}_n \in \mathscr{C}_n$. 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* $\text{Ack}_{n+1} \notin \mathscr{C}_n$.

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

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 $\{\text{Ack}(n,x) \mid 0 \leq n \leq 3,\ 0 \leq x \leq 3\}$, 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

$$\forall n \in \mathbb{N}\ \forall x > 0\ \text{Ack}_{n+1}(x) = \underbrace{\text{Ack}_n \circ \cdots \circ \text{Ack}_n}_{x}(\text{Ack}_{n+1}(0))$$

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 :

$$\forall x \geq 2 \ \forall n \in \mathbb{N} \ \mathrm{Ack}(n, x) \leq \mathrm{Ack}(n+1, x) \,.$$

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

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 récursives

Revenons à la fonction d'évaluation des fonctions récursives primitives. La méthode décrite informellement pour montrer qu'elle 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 que si une fonction partielle f n'est pas définie en une valeur x donnée, c'est que le calcul de $f(x)$ ne termine pas. Dans les langages de programmation usuels cela peut arriver en utilisant une boucle `while`.

Par ailleurs, et bien que cela n'ait pas directement à voir avec les fonctions récursives, il faut remarquer que l'activité des ordinateurs ne se résume pas, très loin de là, au calcul de fonctions (seule chose que formalise la calculabilité), et qu'un programme qui 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.

Le schéma de minimisation Pour étendre 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 définie en dehors), d'autre part ajouter 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 Soit f une fonction récursive partielle. On définit le schéma (ou opérateur) μ par :

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

Le schéma ci-dessus définit intrinsèquement une fonction partielle même si la fonction f est récursive primitive (donc totale). En effet, si $f(\overline{x}, y) \neq 0$ pour toute valeur de $\overline{x}$ et y, 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 z doit être calculable (dans un sens intuitif pour l'instant). 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 $f(\overline{x}, y) = 0$. Chacune des valeurs intermédiaires est définie et le calcul arrive à son terme.

Imaginons la version suivante du schéma de minimisation :

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

Une telle définition est un peu « bancale » : 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'existera pas de procédure évidente pour calculer z. Le fait que la fonction soit définie pour chaque valeur intermédiaire est important.

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 récursives totales f telles que :

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

Une telle fonction f sera dite *régulière*.

Définition 1.2.4 (fonction partielle récursive) L'ensemble des *fonctions partielles 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 récursives partielles, alors $f : \mathbb{N}^n \longrightarrow \mathbb{N}$ définie ci-dessous est une fonction récursive partielle.

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

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 récursives partielles, alors $f : \mathbb{N}^{p+1} \longrightarrow \mathbb{N}$ définie ci-dessous est une fonction récursive partielle.

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

iv. clos par le schéma de minimisation μ.

Par exemple la fonction nulle part définie $x \mapsto \mu t.0 = 1$ est récursive partielle, 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, mais la condition de régularité ne peut se vérifier de façon mécanique, comme on le verra ensuite.

Définition 1.2.5 (fonction récursive totale) L'ensemble des *fonctions récursives totales* 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 récursives partielles. En effet, comme il s'agit d'une définition sur les fonctions totales, la condition de régularité doit être vérifiée pour chaque utilisation du schéma de minimisation, c'est à dire que l'ensembles des fonctions récursives totales est le plus petit sous-ensemble de $\mathscr{F}$

- clos par opérations récursives primitives ;
- tel que si f est récursive totale, f d'arité $k+1$ et f régulière, alors $\overline{x} \mapsto \mu y \cdot \big(f(\overline{x}, y) = 0\big)$ est récursive totale.

Or la condition de régularité n'est pas, contrairement aux autres, une condition purement « syntaxique », vérifiable mécaniquement. On le verra de façon précise ultérieurement, mais on peut le comprendre vu les arguments développés en début de paragraphe (page 14) : c'est la première des deux conditions qui n'est plus vérifiée.

Clairement toute fonction récursive totale est une fonction partielle récursive (qui est totale). Par composition on peut définir des fonctions partielles récursives qui s'avèrent totales à partir de fonctions récursives partielles, comme par exemple en composant une fonction définie seulement en 0 et la fonction nulle. La réciproque n'est donc pas évidente, elle sera obtenue par codage du calcul à la section suivante (voir proposition 1.5.13 page 30).

Définition 1.2.6 (prédicat récursif) Un prédicat d'arité p sur les entiers, un sous-ensemble de $\mathbb{N}^p$, est *récursif* si sa fonction caractéristique est récursive totale.

Noter qu'une fonction caractéristique est toujours totale.

l'ensemble des fonctions récursives totales étant clos par opérations récursives primitives, il contient les fonctions récursives primitives, mais il a également les propriétés de clôture vues sur ces fonctions, et par conséquent l'ensemble des prédicats récursifs a aussi les propriétés de clôture des prédicats récursifs primitifs.

De façon analogue aux fonctions régulères, on appele *prédicat régulier* un prédicat P d'arité $p+1$ récursif et vérifiant

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

et l'ensemble des fonction récursives totales est clos par schéma de minimisation pour les prédicats régulier, 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 $\overline{x} \mapsto \mu y \cdot P\,\overline{x}\,y$, est récursive totale.

On aimerait disposer pour les fonctions récursives partielles de propriétés et schémas de clôture analogues à ceux obtenus pour les fonctions récursives totales, comme le suivant. Une contrainte importante au sujet des fonctions récursives partielles 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, comme le montre l'exemple suivant.

Proposition 1.2.7 *Si P est un prédicat récursif d'arité p, et f et g deux fonctions récursives partielles d'arité p, alors la fonction partielle h définie ci-dessous est 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

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

La proposition se démontrera facilement après avoir introduit une autre caractérisation des fonctions partielles récursives à la section suivante.

En effet le formalisme 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 le voit à la section suivante, dans le cas particulier des machines à registre.

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, qui renvoie à l'instruction qui doit être exécutée. 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. C'est un entier inférieur à la longueur du programme, les instructions étant numérotées de 1 en 1 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 :

$$R_i := R_i - 1$$

3. exécuter un goto conditionnel, si le registre numéro i est nul, aller à l'instruction numéro p :

$$\text{if } R_i = 0 \text{ goto } p$$

4. une instruction d'arrêt qui apparait une et une seule fois en fin du programme :

$$\text{halt}$$

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 $k \geq n$.*

Démonstration. En effet, si $k < n$, il suffit d'ajouter les registres manquant $R_{k+1}, \ldots, R_n$, 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, $\lambda x_1 \ldots x_n.0$.

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

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 R1 = 0 goto 4
1  R1 := R1 − 1
2  R0 := R0 + 1
3  if R3 = 0 goto 0
4  if R2 = 0 goto 8
5  R2 := R2 − 1
6  R0 := R0 + 1
7  if R3 = 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 qui simule 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 :

l	if $R_i = 0$ goto $l+3$
$l+1$	$R_i := R_i - 1$
$l+2$	goto l

   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 :

l	$R_i := 0$
$l+1$	if $R_j = 0$ goto $l+6$
$l+2$	$R_j := R_j - 1$
$l+3$	$R_i := R_i + 1$
$l+4$	$R_{k+1} := R_{k+1} + 1$
$l+5$	goto $l+1$
$l+6$	if $R_{k+1} = 0$ goto $l+10$
$l+7$	$R_{k+1} := R_{k+1} - 1$
$l+8$	$R_j := R_j + 1$
$l+9$	goto $l+6$

et dans le reste du programme on décale de 8 les goto l' pour $l' > l$. ■

Par exemple, avec ces nouvelles instructions, la projection p_n^i est calculée par une machine à n registres de programme

$R_0 := R_i$
halt

donc par une machine à registres avec programme goto.

La fonction successeur est calculée par la machine à 2 registres de programme :

$R_0 := R_1$
$R_0 := R_0 + 1$
halt

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 :
   begin S_1; … ; S_p end
3. une instruction "while", une boucle qui répète une instruction S donnée
   while $R_i \neq 0$ 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'intruction 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 registre 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 séquence d'instruction 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 :

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

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 registre ou sont simulables d'après la proposition 1.3.3.
2. On suppose que les instructions $S_1, \ldots, S_p$ sont simulées par les suites d'instructions $\mathscr{S}_1, \ldots, \mathscr{S}_p$, appelons $\mathscr{S}$ la suite obtenue en les concaténant. Alors la séquence

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

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

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

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

$$\begin{array}{ll}
\vdots & \vdots \\
l & \text{if } R_i = 0 \text{ goto } l+(s+2) \\
\vdots & \mathscr{S} \\
l+s+1 & \text{goto } l \\
l+s+2 & \ldots \\
\vdots & \vdots
\end{array}$$

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 qui est en fait celle de toutes les fonctions récursives partielles (cette équivalence apparait aussi comme conséquence d'un résultat ultérieur). On se sert d'abord du petit résultat 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*

$$\text{if } R_i = 0 \text{ then } \mathscr{S}_1 \text{ else } \mathscr{S}_2$$

qui exécute les instructions $\mathscr{S}_1$ ou $\mathscr{S}_2$ suivant le résultat du test à zéro de R_i. Alors f est calculable par une machine à registres avec programme structuré.

Démonstration. Soit M calculant f et utilisant k registres. On a besoin de deux nouveaux registres que l'on appelera C_1 et C_2. L'instruction

$$\text{if } R_i = 0 \text{ then } \mathscr{S}_1 \text{ else } \mathscr{S}_2$$

est simplement remplacée par

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

■

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 $S_1, \ldots, S_p$. 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 $I_1, \ldots, I_p$ 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.

$$\begin{array}{l}
I_1 := 0; I_2 := 1; \ldots; I_p := 1 \\
\text{while } I_p \neq 0 \text{ do} \\
\quad \text{if } I_1 = 0 \text{ then } \tilde{S}_1 \\
\quad \text{else if } I_2 = 0 \text{ then } \tilde{S}_2 \\
\quad \vdots \\
\quad \text{else if } I_{p-1} = 0 \text{ then } \tilde{S}_{p-1}
\end{array}$$

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 de la preuve « dépend » assez étroitement du programme simulé. On verra plus tard un résultat beaucoup plus fort.

Le lemme suivant sera commode pour montrer que les fonctions 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 :

$$
\begin{array}{l}
R'_1 := R_1 \\
\vdots \\
R'_k := R_k \\
S_1[R'_i/R_i] \\
\vdots \\
S_s[R'_s/R_s] \\
R'_1 := 0 \\
\vdots \\
R'_k = 0
\end{array}
$$

■

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

   $$\text{repeat } S \text{ until } R_i = 0$$

   par un programme goto.

1.4 Les fonctions récursives sont calculables par machines.

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

1.4.1 Les fonctions récursives partielles.

Proposition 1.4.1 *Les fonctions récursives partielles sont calculables par machines à registres avec programme structuré. Par conséquent les fonctions récursives sont calculables sur machines à registres avec programme* while *et programme* goto.

Démonstration. On procède par induction sur la définition des fonctions récursives partielles. On a déjà traité en exemple les fonctions de base, la fonction nulle $\odot$, 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 $g_1, \ldots, g_k : \mathbb{N}^n \to \mathbb{N}$ et $h : \mathbb{N}^k \to \mathbb{N}$ sont calculables par des machines $M_1, \ldots, M_k$ et M_0, dont les programmes sont les suites d'instructions $\mathscr{S}_1, \ldots, \mathscr{S}_k$ 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 $R_0, R_1, \ldots, R_m, H_1, \ldots, H_k$. Elle va calculer successivement $g_1(x_1, \ldots, x_n)$,. . .,$g_k(x_1, \ldots, x_n)$ dont elle va stocker les valeurs dans les registres $H_1, \ldots, H_k$, puis calculer $f(H_1, \ldots, H_k)$. Voici le programme de cette machine

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

Remarquez que dès que l'une des machines $M_1, \ldots, M_k$ 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 $g : \mathbb{N}^n \to \mathbb{N}$ $(n \leq k_1 + 1)$. Soit M_2 une machine propre à k_2 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 :

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

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 :

$\mathscr{S}_0$
$R_0 := R_0 + 1$
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 1.4.2 *Les fonctions récursives primitives sont calculables par programme* for.

La réciproque de ce résultat est vraie.

Proposition 1.4.3 *Toute fonction calculable par programme* for *est récursive primitive.*

Il faut bien remarquer que les boucles for que nous avons défini 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 Démontrer la proposition précédente : 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 récursives.

Nous allons maintenant montrer que les fonctions partielles calculables par machine goto sont récursives. Pour cela on va coder par des entiers les machines elles-mêmes, et les états possibles d'une machine à un instant donné. Ensuite on montrera que le calcul de la machine sur un nombre t d'étapes, vue comme le calcul du code de son état au bout de t étapes, se code de façon récursive.

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'un 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. on va commencer par coder ses instructions.

Préliminaires et notations sur les fonctions récursives primitives.

On va utiliser les différents codages primitif récursif des couples, k-uplets et suite finies définis précédemment, ainsi que des fonctions usuelles sur les suites. Les arguments de la fonction $nth(l,i)$ qui calcule le i-ème élément de la suite de code l sont notés dans cet ordre.

Pour simplifier les notations on note ainsi les bijections de Cantor :

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

On rappelle que par définition des α_i :

$$\langle x_1, \langle x_2, \ldots, x_k\rangle\rangle = \langle x_1, \ldots, x_k\rangle \ .$$

et que les projections vérifient $\pi_1 = \pi_1^2 = \pi_1^3 = \ldots$ 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 :

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

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

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

Codage des instructions.

On a 4 sortes d'instructions, chacune pouvant être paramétrée par un entier (le numéro de registre pour l'incrémentation et la décrémentation), ou un couple d'entier (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 :

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

De fait, le codage n'a pas besoin d'être fonctionnel, et nous considérerons 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 vraiment besoin du résultat de l'exercice suivant.

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

Codage de l'état d'une machine.

On code l'état e 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. Il nous faut essentiellement pour coder le calcul les deux fonctions et le prédicat suivants :

- Une fonction d'initialisation $\mathsf{init}_n : \mathbb{N}^{n+1} \to \mathbb{N}$. 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 $\mathsf{tr} : \mathbb{N}^2 \to \mathbb{N}$. Cette fonction calcule en fonction du code m d'une machine et de l'état e de cette machine l'état de la machine après une étape de calcul à partir de e.

– Un prédicat Halt de terminaison à deux arguments qui 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 que l'on peut trouver de telles fonctions et prédicat calculables, donc récursifs d'après la thèse de Church. On montre explicitement dans les sections qui suivent que l'on peut trouver de telles fonctions et prédicat qui sont primitifs récursifs.

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 $k < n$) :

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

Lemme 1.5.1 *la fonction* $\mathrm{init}_n(m, x_1, \ldots, x_n)$ *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 primitif récursif. 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 10), on a :

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

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 $e = \langle i,r\rangle$:

si $\pi^2_1(\mathrm{inst}(m,\langle i,r\rangle)) \in \{0,1\}$ — si incrémentation ou décrémentation
  alors $\mathrm{tr}_1(m,\langle i,r\rangle) = i+1$ — on passe à l'instruction suivante
  sinon si $\pi^2_1(\mathrm{inst}(m,\langle i,r\rangle)) = 2$ — si goto
    alors si $\mathrm{nth}(r,\pi^3_2(\mathrm{inst}(m,\langle i,r\rangle))) = 0$ — si $R_j = 0$
      alors $\mathrm{tr}_1(m,\langle i,r\rangle) = \pi^3_3(\mathrm{inst}(m,\langle i,r\rangle))$ — aller à la ligne indiquée
      sinon $\mathrm{tr}_1(m,\langle i,r\rangle) = i+1$ — sinon on passe à l'instruction suivante
    sinon $\mathrm{tr}_1(m,\langle i,r\rangle) = i$ — sinon (halt) arrêt

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

si $\pi^2_1(\mathrm{inst}(m,\langle i,r\rangle)) = 0$ — si incrémentation
  alors $\mathrm{tr}_2(m,\langle i,r\rangle) = \mathrm{inc}(\pi^2_2(\mathrm{inst}(m,\langle i,r\rangle)),r)$ — $R_j := R_j+1$
  sinon si $\pi^2_1(\mathrm{inst}(m,\langle i,r\rangle)) = 1$ — si décrémentation
    alors $\mathrm{tr}_2(m,\langle i,r\rangle) = \mathrm{dec}(\pi^2_2(\mathrm{inst}(m,\langle i,r\rangle)),r)$ — $R_j := R_j-1$
    sinon $\mathrm{tr}_2(m,\langle i,r\rangle) = r$ — sinon (goto ou halt) les registres ne sont pas modifiés

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 :

$$\mathrm{Halt}(m,e) \equiv \pi^2_1(\mathrm{inst}(m,e)) > 2$$

Lemme 1.5.4 *Le prédicat binaire* Halt *défini ci-dessus est primitif récursif. 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 calculables par une classe plus étendue (au moins en apparence) de machines sont récursives partielles. Nous allons décrire leur comportement, ce qui ne sera vraiment utile qu'au paragraphe 1.5.5.

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 $\pi^2_1(c) > 2$ 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 intruction 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.

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 le temps de calcul, c'est à dire le nombre d'étapes jusqu'à ce que la machine termine, par minimisation.

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

$$\mathrm{st}_n(m, x_1, \ldots, x_n, 0) = \mathrm{init}_n(m, x_1, \ldots, x_n)$$
$$\mathrm{st}_n(m, x_1, \ldots, x_n, t+1) = \mathrm{tr}(m, \mathrm{st}_n(m, x_1, \ldots, x_n, t))$$

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

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

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

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

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 :

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

Elle est récursive partielle. Remarquez que la fonction partielle calculée est de toute façon récursive, que m soit ou non le 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 primitif récursif H_n à $n+2$ arguments tels que, si m est le code d'une machine M 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 n-aire f calculée par M est récursive partielle 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 calculable par machine à registres avec programme* goto*, avec programme structuré, ou avec programme* while *est récursive partielle.*

Corollaire 1.5.7 *Toute fonction récursive partielle s'écrit comme composée d'une fonction récursive primitive et d'une fonction définie par minimisation sur un prédicat primitif récursif.*

On déduit du corollaire précédent que l'on obtient toutes les fonctions récursives partielles en restreignant dans la définition le schéma de minimisation $\mu t.[g(\overline{x}, t) = 0]$ aux fonctions g totales.

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 primitifs récursifs. ■

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 $R_i := 0$ 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 recursives définissent la même notion de fonction calculable. La méthode utilisée pour montrer que les fonctions partielles calculables par machine sont 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, 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 récursives (totales), et que le prédicat d'arrêt est récursif.

Ainsi, sans grande surprise, on a le résultat suivant [3].

Proposition 1.5.10 *Une fonction $f : \mathbb{N}^p \to \mathbb{N}$ est calculable par machine de Turing si et seulement si elle est récursive.*

Bien d'autres modélisations de la notion de fonction calculable existent. On citera entre autres 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. A nouveau, tous ces modèles sont équivalents du point de vue de la calculabilité. Cette série de résultats semble confirmer la conjecture suivante, dite thèse de Church.

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

La méthode mise en œuvre ici pour les machines à registres parait très générale, ce est aussi un argument pour cette thèse.

Cependant cette affirmation n'est pas démontrable : elle met en relation une notion intuitive (le fait d'être calculable) avec une notion mathématiquement précise (être calculable dans un modèle particulier). En dépit de nombreux efforts, elle n'a pas non plus été infirmée.

3. Les détails sont laissés au lecteur, mais tous les ingrédients se trouvent dans les preuves des résultats de simulation vus précédemment.

1.5.4 Forme normale de Kleene.

On va donner une démonstration alternative des résultats précédents qui fournit une forme un peu plus simple pour les fonctions calculables (on va préciser le corollaire 1.5.7), en faisant intervenir le code de la suite des états de la machine, plutôt que le temps de calcul (qui est la longueur de cette suite).

Proposition 1.5.11 *Pour chaque entier $n \geq 1$ il existe un prédicat primitif récursif noté T^n et une fonction récursive primitive $U : \mathbb{N} \to \mathbb{N}$ tels que pour toute machine de code m calculant une fonction à n arguments f :*

1. $\exists s\, T^n(m, x_1, \ldots, x_n, s) \Leftrightarrow f(x_1, \ldots, x_n) \downarrow$
2. $T^n(m, x_1, \ldots, x_n, s) \Rightarrow U(s) = f(x_1, \ldots, x_n)$
3. $\big(T^n(m, x_1, \ldots, x_n, s) \text{ et } T^n(m, x_1, \ldots, x_n, s')\big) \Rightarrow s = s'$

Démonstration. L'idée est que s désigne le code (en commençant par la fin) d'une suite d'états correcte pour m, soit $[e_n; \ldots; e_0]$. On commence par définir un prédicat P_n qui indique que la suite d'états est bien correcte (sans que le calcul soit forcément fini), pour une machine de code m avec $x_1, \ldots, x_n$ en entrées. Le prédicat P_n est défini par sa fonction caractéristique f_n :

$$\begin{aligned}
&f_n(m, x_1, \ldots, x_n, []) = 1\\
&f_n(m, x_1, \ldots, x_n, [e]) = \chi_=(\mathrm{init}_n(m, x_1, \ldots, x_n), e)\\
&f_n(m, x_1, \ldots, x_n, e :: s) = f_n(m, x_1, \ldots, x_n, s) \cdot \chi_=(\mathrm{tr}(m, \mathrm{hd}(s)), e)
\end{aligned}$$

la fonction f est définie par récurrence structurelle sur les listes, à partir de fonctions récursives primitives. on a vu qu'alors f est récursive primitive. Le prédicat P_n est donc primitif récursif. On définit ensuite

$$\begin{aligned}
&T'^n(m, x_1, \ldots, x_n, s) \equiv P_n(m, x_1, \ldots, x_n, s) \text{ et } \mathrm{Halt}(m, \mathrm{hd}(s))\\
&T^n(m, x_1, \ldots, x_n, s) \equiv T'^n(m, x_1, \ldots, x_n, s) \text{ et } \forall s' < s \neg T'^n(m, x_1, \ldots, x_n, s')
\end{aligned}$$

Le prédicat primitif récursif T'^n ne vérifie a priori que la première condition de la proposition (terminaison). Le prédicat T^n est primitif récursif et vérifie les deux premières conditions : terminaison et fonctionalité pour s en sortie.

La fonction U a besoin d'extraire la valeur de R_0 du code du premier état de s soit :

$$U(s) = \mathrm{hd}(\pi_1^2(\mathrm{hd}(s)))$$

fonction qui est bien récursive primitive. ■

Corollaire 1.5.12 (forme normale de Kleene) *Les prédicats T^n et la fonction U sont ceux de la proposition précédente. Pour toute fonction récursive $f : \mathbb{N}^n \to \mathbb{N}$, il existe au moins un entier m tel que :*

$$f(x_1, \ldots, x_n) = U(\mu s. T^n(m, x_1, \ldots, x_n, s))\ .$$

Démonstration. Il suffit de prendre le code m d'une machine qui calcule f. ■

Une expression de la fonction f comme celle donnée dans le corollaire :

$$f(x_1, \ldots, x_n) = g(\mu s. P(x_1, \ldots, x_n, s))$$

où $g : \mathbb{N} \to \mathbb{N}$ est une fonction récursive primitive et P un prédicat primitif récursif est dite sous *forme normale de Kleene.*

On peut se demander si une fonction récursive partielle qui s'avère être totale peut être définie directement par les moyens utilisés pour les fonctions récursives totales (i.e. la minimisation sur les fonctions ou les prédicats *réguliers*). Le résultat suivant répond par l'affirmative.

Proposition 1.5.13 *L'ensemble des fonctions récursives totales est égal à l'ensemble des fonctions récursives partielles qui s'avèrent être totales.*

Démonstration. Si f est une fonction récursive totale, elle est évidemment aussi une fonction récursive partielle totale. Réciproquement, si f est une fonction récursive partielle, alors pour un certain $m \in \mathbb{N}$:

$$f(x_1, \ldots, x_n) = U(\mu s.T^n(m, x_1, \ldots, x_n, s)) \,.$$

Si f est une fonction totale, pour tout $x_1, ..., x_n$, $f(x_1, ..., x_n)$ est définie et donc, il existe s tel que $T^n(m, x_1, \ldots, x_n, s)$. Ceci implique que $T_n(m, x_1, \ldots, x_n, s)$ est régulier. Comme f est définie par composition à partir de fonctions récursives primitives et d'une fonction définie par minimisation à partir d'un prédicat régulier, f est une fonction récursive totale. ∎

1.5.5 Fonctions universelles.

Propriété d'énumération.

Pour chaque $n \geq 1$ on définit la fonction d'arité $n+1$ φ^n :

$$\varphi^n(m, x_1, \ldots, x_n) = U(\mu s.T^n(m, x_1, \ldots, x_n, s))$$

et on note simplement φ pour φ^1.

Cette fonction récursive partielle permet de calculer toutes les fonctions récursives partielles à n arguments. On dit que c'est une *fonction universelle* pour les fonctions récursives partielles à n arguments. On peut réécrire ainsi le corollaire précédent :

Théorème 1.5.14 (Théorème d'énumération) *la famille de fonctions $(\varphi^n)_{n \in \mathbb{N}^*}$, définie ci-dessus est telle que pour toute fonction récursive partielle $f : \mathbb{N}^n \to \mathbb{N}$ il existe au moins un entier m tel que :*

$$f(x_1, \ldots, x_n) = \varphi^n(m, x_1, \ldots, x_n)$$

Un tel entier m sera appelé un *indice de f* (rappelons que c'est le code d'une machine qui calcule f). On notera $f = \varphi^n_m$ si m est un code de f.

En fait chacune des fonctions $(\varphi^n)_{n \in \mathbb{N}^*}$ est universelle en un sens un peu plus fort. Par exemple, la fonction $\varphi = \varphi^1$ énumère modulo codage toutes les fonctions récursives partielles :

Corollaire 1.5.15 *Pour toute fonction récursive partielle $f : \mathbb{N}^n \to \mathbb{N}$, il existe un entier i tel que :*

$$f(x_1, \ldots, x_n) = \varphi(i, \langle x_1, \ldots, x_n \rangle)$$

Démonstration. On prend pour i un indice de la fonction $\lambda x.f(\pi^n_1(x), \ldots, \pi^n_n(x))$. ∎

Il est intuitivement clair (voir exercice suivant) qu'une fonction a une infinité d'indices : on peut toujours ajouter des détours inutiles dans le programme d'une machine. On montrera plus tard que c'est en un certain sens inévitable.

Exercice 14 (Lemme de bourrage)

1. étant donné une machine M à k registres, construire une machine M', avec autant de registres, qui calcule la même fonction, et dont le programme a un code strictement plus grand que celui de M.
2. Montrer qu'il existe une fonction récursive primitive u_n telle que

$$\varphi^n_{u_n(m)} = \varphi^n_m \text{ et pour tout } m \; u_n(m) > m$$

3. Montrer qu'il existe une fonction récursive primitive $h_n(p, m)$ telle que h_p est strictement croissante en la première variable et pour tout entier m, $\varphi^n_{h(p,m)} = \varphi^n_m$.

Une conséquence immédiate du théorème d'énumération est que, la fonction φ^n étant récursive partielle à $n+1$ arguments, elle est a un indice : le code d'une machine qui la calcule, que l'on appelle *machine universelle.*

Corollaire 1.5.16 *Pour tout $n \geq 1$ il existe un entier m_n tel que :*

$$\varphi^n(m, x_1, \ldots, x_n) = \varphi^{n+1}(m_n, m, x_1, \ldots, x_n) \,.$$

Propriété de paramétrisation.

Une propriété en quelque sorte réciproque de celle que l'on vient d'énoncer, et caractéristique des familles de fonctions universelles, est la suivante.

Théorème 1.5.17 (théorème s_n^m) *Pour tous entiers $m, n \geq 1$ il existe une fonction récursive totale $s_n^m : \mathbb{N}^{m+1} \to \mathbb{N}$ telle que :*

$$\varphi^{m+n}(i, x_1, \ldots, x_m, y_1, \ldots, y_n) = \varphi^m(s_n^m(i, x_1, \ldots, x_m), y_1, \ldots, y_n)$$

Pour la famille de fonctions universelle étudiée ici, la fonction s_n^m est récursive primitive.

Si on on revient aux machines, la fonction s_n^m permet donc, à partir du code i d'une machine M à $m+n$ entrées, et de m entiers $x_1, \ldots, x_m$, de calculer le code d'une machine M' à n entrées, qui fonctionne comme M, après avoir fixé les m premières entrées à $x_1, \ldots, x_m$.

Démonstration. On va suivre la remarque précédente. Il s'agit d'abord d'indiquer comme construire M', puis on montre que les constructions se codent de façon récursive primitive. Il faut veiller à ce que la fonction calculée soit bien celle cherchée même quand l'indice $i = \langle k, l \rangle$ n'est pas un « vrai » code de machine (cf. paragraphe 1.5.2). La machine M' va avoir les mêmes k registres. On construit son programme à partir de celui de M en 3 étapes.

1. On ajoute en tête de programme des instructions qui essentiellement (c'est-à-dire quand il y a suffisamment de registres) recopient les registres d'entrées R_1, …, R_n à partir de R_{m+1} (registres de travail), et mettent à 0 les n premiers registres d'entrées). il faut en fait discuter suivant le nombre de registres k. L'opération est donnée,
   si $k \leq m$ par la séquence
   $$R_k := 0, \ldots, R_1 := 0$$
   si $m < k < m+n$ par la séquence
   $$R_k := R_{k-m}, \ldots, R_{m+1} := R_1$$
   et si $k \geq m+n$ par la séquence
   $$R_{m+n} := R_n, \ldots, R_{m+1} := R_1 \ .$$
   Chacune des instructions $R_{m+i} := R_i$, respectivement $R_i := 0$, s'écrit

l	if $R_i = 0$ goto $l+4$	l	if $R_i = 0$ goto $l+3$
$l+1$	$R_{m+i} := R_{m+i} + 1$	$l+1$	$R_i := R_i - 1$
$l+2$	$R_i := R_i - 1$	$l+2$	goto l
$l+3$	goto l		

   avec, par exemple dans le dernier cas $l = 4(n-i)$. On ajoute $3k$ lignes dans le premier cas, $4(k-m)$ dans le second cas et $4n$ lignes dans le dernier.
2. On ajoute ensuite les instructions qui correspondent à :
   $$R_1 := x_1, \ldots, R_m := x_m$$
   chaque opération $R_i := x_i$ s'écrit (le registre R_i est à 0) :
   $$\underbrace{R_i := R_i + 1, \ldots, R_i := R_i + 1}_{x_i}$$
   on ajoute donc $x_1 + \cdots + x_m$ lignes.
3. On ajoute le programme de M modifié de façon à se comporter de la même façon que pour la machine M, c'est-à-dire que les instructions goto doivent pointer sur les numéros de lignes décalées du nombre d'instructions ajouté en tête de programme. Ce nombre est une fonction de $(k, x_1, \ldots, x_m)$ (m et n sont des constantes).

Il est à peu près clair qu'à chaque étape correspond une opération récursive primitive sur le code. Ceux qui n'en sont pas convaincus sont invités à le démontrer eux-mêmes, ou, en dernier ressort, à chercher de possibles erreurs dans les détails donnés ci-dessous.

1. la fonction $k, s \mapsto f_1(k, s)$ qui ajoute le code de la liste des instructions de l'étape 1 et la fonction $k \mapsto nb(k)$ qui donne le nombre d'instruction ajoutées, sont récursives primitives. Elles se définissent par cas et récurrence sur k comme indiqué.

2. Le code d'une suite de x incrémentations sur le registre i se définit par récurrence récursive primitive sur x

$$a(i,x) = []$$
$$a(i,x+1) = \langle 0,i\rangle :: a(i,x) \quad \text{0 est le code de l'incrémentation}$$

La fonction a est récursive primitive, et donc également la fonction $f_2(s, x_1, \ldots, x_m)$ qui ajoute la liste d'instructions souhaitée en tête du programme s (m est une constante) :

$$f_2(x_1,\ldots,x_m,s) = a(1,x_1)@\ldots@a(n,x_m)@s$$

3. La fonction $d(g,e)$ décale de g les numéro de ligne dans le code d'une instruction e s'il s'agit d'un goto, et ne fait rien sinon. Elle est récursive primitive.

si $\pi^2_1(e)$ si goto
alors $d(g,e) = \langle 2, \pi^3_2(e), \pi^3_3(e)+g\rangle$
sinon $d(g,e) = e$

Il suit que, par composition, la fonction $d'(x_1,\ldots,x_m,i,e) = d((nb(\pi^2_1(i)) + x_1 + \cdots + x_m, e)$ est récursive primitive, et donc également La fonction f_3 qui calcule le résultat de l'application de la fonction $e \mapsto d'(x_1,\ldots,x_m,i,e)$ à toutes les instructions du programme ; celui-ci étant une liste on utilise une fonction map, (voir page 10), soit $f_3(x_1,\ldots,x_m,i) = \mathrm{map}_{d'}(\pi^2_2(i))$. Cette dernière construit à partir du code i de la machine M le code du programme avec les renumérotations de registres et décalages des goto souhaités.

On obtient finalement la fonction s^m_n de façon récursive primitive :

$$s^m_n(i,x_1,\ldots,x_m) = \langle \pi^2_1(i), f_1(\pi^2_1(i), f_2(x_1,\ldots,x_m, f_3(i,x_1,\ldots,x_m)))\rangle \qquad \blacksquare$$

La famille de fonctions universelles telle qu'elle est construite est très arbitraire : elle dépend non seulement du choix des machines à registres pour représenter le calcul, mais aussi du choix du codage qui n'a rien d'intrinsèque.

On verra plus tard que la propriété d'énumération et l'existence de fonctions s^m_n, appelée aussi propriété de paramétrisation, axiomatisent les familles de fonctions récursives partielles universelles : toute famille de fonctions universelles $(\psi^n)_{n\geq 1}$ qui vérifient ces deux propriétés se déduit de $(\varphi^n)_{n\geq 1}$ par une permutation récursive h :

$$\psi^1_i = \varphi^1_{h(i)} .$$

L'importance de la propriété de paramétrisation va apparaître dans les chapitres qui suivent. Avec les fonctions universelles, les entiers jouent un double rôle : celui de donnée, mais aussi celui de programme. La fonction s^m_n permet donc également de décrire des opérations calculables sur les programmes et de composer ceux-ci. Voyons un exemple simple. Soit e le code de la fonction $(i,j,x) \mapsto \varphi^1_i(x) + \varphi^1_j(x)$:

$$\varphi(i,x) + \varphi(j,x) = \varphi^3(e,i,j,x) = \varphi^1(s^2_1(e,i,j),x) .$$

La fonction $(i,j) \mapsto s^2_1(e,i,j)$ calcule donc, à partir des indices de deux fonctions, l'indice de la somme de ces deux fonctions. Il fabrique en quelque sorte le (code du) programme pour la somme de ces deux fonctions, en intégrant comme sous-programmes les programmes (de codes i et j) calculant chacune de ces deux fonctions.

1.6 Problèmes indécidables

1.6.1 Ensembles récursivement énumérables

Définition 1.6.1 (ensemble récursivement énumérable) Un sous-ensemble A de $\mathbb{N}^k$ est dit *récursivement énumérable* quand il est le domaine d'une fonction récursive (partielle).

On parle aussi de *prédicat récursivement énumérable* pour un prédicat d'arité k dont le support est récursivement énumérable. On utilisera parfois l'abréviation r.e..

On notex W_m^k le domaine de définition de la fonction (à k variables) d'indice m, φ_m^k. L'ensemble des prédicats récursivements énumérables est donc constitué des ensembles W_m^k.

Notons tout d'abord qu'il est toujours possible de se ramener aux sous-ensembles de $\mathbb{N}$ via codage.

Lemme 1.6.2 *Soit A un sous-ensemble de $\mathbb{N}^k$, alors :*
- *A est r.e. si et seulement si $\{\langle x_1,\dots,x_k\rangle \mid (x_1,\dots,x_k)\in A\}$ est r.e. ;*
- *A est récursif si et seulement si $\{\langle x_1,\dots,x_k\rangle \mid (x_1,\dots,x_k)\in A\}$ est récursif.*

On rappelle que le résultat est encore vrai en prenant primitif récursif pour récursif.

Démonstration. Par composition avec α_k pour un sens, $(\pi_1^k,\dots,\pi_k^k)$ pour l'autre. ■

Proposition 1.6.3 (caractérisation des ensembles récursivement énumérables) *Soit $A \subseteq \mathbb{N}^k$. Alors A est récursivement énumérable dès qu'il satisfait l'une des assertions suivantes qui sont équivalentes :*

1. *A est le domaine d'une fonction récursive partielle ;*
2. *A est le projeté d'un ensemble récursif primitif, B, c'est-à-dire que $A = \{\overline{x} : \exists y(\overline{x}, y)\in B\}$;*
3. *A est le projeté d'un ensemble récursif ;*
4. *A est l'image d'une fonction récursive partielle, modulo codage des k-uplets, soit il existe f récursive partielle telle que* $\mathrm{Im}\, f = \{\langle x_1,\dots,x_k\rangle \mid (x_1,\dots,x_k)\in A\}$.
5. *A est vide ou l'image d'une fonction récursive primitive à un argument modulo codage des k-uplets, soit il existe f fonction récursive primitive telle que* $\mathrm{Im}\, f = \{\langle x_1,\dots,x_k\rangle \mid (x_1,\dots,x_k)\in A\}$ *;*
6. *A est vide ou l'image d'une fonction récursive totale à un argument, modulo codage des k-uplets.*

La dernière caractérisation justifie l'appellation « récursivement énumérable ». Quand $A\subseteq\mathbb{N}$ elle s'écrit plus simplement

6'. A est vide où il existe une fonction récursive totale f telle que $A = \mathrm{Im}\, f$.

Démonstration. On peut supposer par codage (lemme 1.6.2) que A est un sous-ensemble de $\mathbb{N}$.

(1 ⇒ 2) : soit φ_m la fonction dont A est le domaine, alors $\left\{(x,s) \mid T^1(m,x,s)\right\}$ convient, où T^1 est le prédicat d'arrêt de Kleene, qui est récursif primitif, cf. proposition 1.5.11.

(2 ⇒ 3) : évident.

(3 ⇒ 6) et (2 ⇒ 5) : on suppose A non vide, soit donc $a \in A$. La fonction f qui à $z = \langle x, y\rangle$ associe x si $(x, y) \in B$, a sinon (c'est une définition par cas de fonction récursive totale, voir page 6), convient.

(6 ⇒ 4) et (5 ⇒ 4) : l'ensemble vide est l'ensemble image de la fonction nulle part définie.

(4 ⇒ 1) : soit φ_m la fonction récursive partielle dont A est l'image. Alors la fonction suivante convient (les notations sont celles de Kleene, cf. proposition 1.5.11)

$$f(y) = \mu t\cdot\left(T^1(m,\pi_1^2(t),\pi_2^2(t)) \text{ et } y = U(\pi_2^2(t))\right) .$$

Elle est bien définie si la machine de code m s'arrête avec y en valeur de sortie. ■

De la caractérisation d'un r.e. comme projeté d'un ensemble récursif, c'est-à-dire que tout prédicat r.e. est obtenu par quantificatione existentielle sur un prédicat récursif, on en déduit la clôture par projection des r.e.. Certaines propriétés de clôture des prédicats récursifs (qui sont clos par opérations primitives récursives donc a les propriétés de clôture du paragraphe 1.1.2) sont héritées par projection, mais pas toutes.

Proposition 1.6.4 (propriétés de clôture des récursivement énumérables) *La classe des prédicats récursivement énumérables est close par*

1. *conjonction et disjonction ;*
2. *quantification existentielle ;*
3. *quantification universelle bornée par une variable.*

Ceci se traduit par le fait que la classe des ensembles récursivement énumérables est close par intersection, produit cartésien, réunion, projection (et l'opération correspondant à la quantification universelle bornée sur une variable).

Démonstration.

1. Soit $P\,x_1,\ldots,x_p,z_1,\ldots,z_k$ et $Q\,y_1,\ldots,y_q,z_1,\ldots,z_k$ deux prédicats r.e., caractérisés par les domaines des fonctions f et g, alors $f+g$ a un domaine caractérisé par la conjonction de ces deux prédicats.
   La disjonction a réellement un sens pour des prédicats de même arité, il suffit de complèter pour que les prédicats portent P et Q sur les mêmes variables $\overline{x}$. Soient A et B des prédicats récursifs tels que
   $$P\overline{x} \equiv \exists z A\overline{x}, z\ ;\ \ Q\overline{x} \equiv \exists z B\overline{x}, z\ ;$$
   On a $(P \wedge Q)\overline{x} \equiv \exists z(A\overline{x}, z \vee B\overline{x}, z)$.
2. On a pour P r.e. A récursif tel que $P\overline{x}, y \equiv \exists z A\overline{x}, y, z$ et donc $\exists y\, P\overline{x}, y \equiv \exists t\, A\overline{x}, \pi_1^2(t), \pi_2^2(t)$, quantification existentielle sur un prédicat récursif par composition.
3. Avec les mêmes notations on a $\forall y \leq t\, P\overline{x}, y \equiv \forall y \leq t\, \exists z\, A\overline{x}, y, z$. Il s'agit déchanger les deux derniers quantificateurs. Soient $\overline{x}, t$ tels que $\forall y \leq t\, P\overline{x}, y$. Nous sommes dans $\mathbb{N}$, il n'y a qu'un nombre fini de $y \leq t$, t étant fixé, pour lesquels ont choisit un témoin z_y tel que $A\overline{x}, y, z_y$. Soit $u = \sup_{y \leq t} z_y$. On a alors $\forall y \leq t\, \exists z \leq u\, A\overline{x}, y, z$. Ce prédicat est récursif et donc le prédicat $\exists u\, \forall y \leq t\, \exists z \leq u\, A\overline{x}, y, z$, qui est satisfait par $\overline{x}, t$, est r.e. Réciproquement si ce prédicat est satisfait apr $\overline{x}, t$ on a évidemment $\forall y \leq t\, P\overline{x}, y$, d'où le résultat cherché. ■

La proposition suivante relie les notions d'ensemble récursif et récursivement énumérable.

Proposition 1.6.5 *Un ensemble $A \subseteq \mathbb{N}^k$ est récursif si et seulement si A et son complémentaire $A^c = \mathbb{N}^k \setminus A$ sont récursivement énumérables.*

Démonstration. Si A est récursif alors clairement A et A^c sont récursivement énumérables. Pour la réciproque, si A et $compA$ sont les domaines (disjoints) respectives de deux fonctions partielles récursives d'indices m_0 et m_1, la fonction s définie par

$$s(\overline{x}) = \mu y.[T^k(m_0, \overline{x}, y) \vee T^k(m_1, \overline{x}, y)]$$

est récursive totale. On a alors : $A(\overline{x})$ si et seulement si $T^k(m_0, \overline{x}, s(\overline{x}))$ et A est donc récursif[4]. On peut examiner l'argument ci-dessus avec un point de vue plus algorithmique. Supposons qu'il existe une machine M_1 ne terminant que sur les éléments de A et une machine M_2 ne terminant que sur les éléments de $\mathbb{N}^k \setminus A$. On a simplement construit une machine M qui, sur une entrée $\overline{x}$ exécute les deux machines M_1 et M_2 sur $\overline{x}$ en parallèle (en entrelaçant les instructions des machines[5]. L'entrée $\overline{x}$ est alors acceptée ou refusée suivant laquelle de M_1 et M_2 s'arrête. Noter qu'on ne peut pas lancer une exécution (celle de M_1, par exemple) puis l'autre car la première exécution peut ne pas terminer. ■

Dès que l'on aura montré l'existence d'un ensemble récursivement énumérable non récursif, ce qui sera fait au paragraphe suivant, on pourra déduire de cette proposition et de la caractérisation des r.e. que

- la classe des ensembles récursivement énumérables n'est pas close par complémentation (contrairement à celle des ensembles récursif) ;

4. vous pouvez pousser les détails jusqu'à donner la définition de la fonction caractéristique de A
5. essayez de donner les détails pour une machine à registres. Aide : faites travailler les machines sur deux ensembles de registres différents. Utiliser 3 nouveaux registres en plus : deux pour garder, à tout instant le numéro de l'instruction de chacun des programmes en cours ; un dernier pour compter les pas de programme et permettre que tout étape impaire corresponde à une étape de simulation de M_1 et toute étape paire pour M_2

- la classe des ensembles récursifs n'est pas close par projection (contrairement à celle des ensembles r.e.).

Exercice 15 Démontrer la proposition 1.2.7 page 16.

1.6.2 Prédicats et problèmes

On peut construire naturellement une notion de problème "algorithmique" à partir de celle de prédicats. Soit $A \subseteq \mathbb{N}^k$, on peut consiérer le problème suivant :

A
Entrée : $\overline{x} \in \mathbb{N}^k$
Question : $\overline{x}$ appartient-t-il à A ?

Lorsque A est un prédicat récursif, on dit alors que le problème associé à A est *décidable*. Il est dit *indécidable* dans le cas contraire. On trouve parfois dans la littérature l'expression *semi-décidable* pour désigner un problème associé à un prédicat récursivement énumérable.

1.6.3 Problème de l'arrêt : la méthode diagonale

On montre dans cette partie que certains problèmes sont indécidables par nature (bien qu'ils puissent être récursivement énumérable). La méthode employée est bien connue des mathématiciens depuis Cantor : il s'agit de la diagonalisation. Pour arriver à notre résultat le plus général, le problème de l'arrêt d'une machine, on commence par le problème suivant.

Théorème 1.6.6 (problème de l'arrêt diagonal) *L'ensemble* $K = \{m : \varphi^1_m(m)\downarrow\}$ *n'est pas récursif.*

Démonstration. Considérons la fonction récursive à deux variables $\varphi^1_m(x) = \varphi^1(m,x)$. Son domaine est récursivement énumérable et donc celui de la fonction f telle que $f(m) = \varphi^1_m(m)$ i.e. l'ensemble K. Supposons que le complémentaire de K, noté K^c soit récursivement énumérable [6]. Dans ce cas, K^c est le domaine d'une certaine fonction g récursive dont l'indice est a i.e. $g = \varphi^1_a$.

Une question naturelle est alors : dans quel ensemble se trouve a ? On voit par définition que :

$$\begin{aligned} a \in K^c &\iff a \in \{x : \varphi^1_a(x)\downarrow\} \\ &\iff \varphi^1_a(a)\downarrow \\ &\iff a \in K \end{aligned}$$

ce qui est une contradiction. Donc K^c n'est pas r.e. et K ne peut être récursif. ■

Le prédicat considéré dans la preuve précédente est associé au problème suivant :

Diag
Entrée : Une machine à registre (ou de Turing) M donnée par son code m
Question : le calcul de M sur l'entrée m s'arrête-t'il ?

Le théorème 1.6.6 dit que le problème Diag est indécidable. Même si cela est redondant avec la preuve déjà donnée, on peut regarder comment l'argument diagonal principal se déploit dans le cas d'une preuve directe de l'indécidabilité de ce problème.

Supposons que le problème Diag est décidable par une machine (mettons à registres avec programme structuré) M. Plus exactement, M calcule la fonction caractéristique $Diag(m) = 1$ si $m \in$ Diag et $Diag(m) = 0$ sinon. On modifie la machine M en une nouvelle machine M' en la faisant boucler indéfiniment sur l'entrée m lorsque $Diag(m) = 1$. Cela peut se faire en ajoutant l'instruction suivante en fin de programme :

while $R_0 \neq 0$ do   end

6. cette assertion est déjà, intuitivement, douteuse : comment énumérer les entrées x sur lesquelles le calcul de l'image de $f(x)$ ne s'arrête pas... ?

On a maintenant une machine M' qui s'arrête sur m si $m \not\in$ DIAG et qui ne s'arrrête pas sinon. Que fait la machine M' sur l'entrée constituée de son propre code m' ? on a :

M' s'arrête sur l'entrée m' ssi $m' \not\in$ DIAG ssi M' ne s'arrête pas sur m' (par définition de DIAG). Ce qui nous amène à une contradiction.

On peut maintenant considérer le problème plus général suivant et montrer, comme conséquence du Théorème 1.6.6, qu'il est indécidable.

ARRET
Entrée : Une machine à registre (resp. de Turing) M, $x \in \mathbb{N}$ (resp. x un mot sur l'alphabet d'entrée Σ de M)
Question : le calcul de M sur l'entrée x s'arrête-t'il ?

Théorème 1.6.7 (problème de l'arrêt) *L'ensemble* $\{(m,x) : \varphi^1(m,x)\downarrow\}$ *n'est pas récursif.*

Démonstration. Si l'ensemble $\{(m,x) : \varphi^1(m,x)\downarrow\}$ était récursif, l'ensemble $\{m : \varphi^1(m,m)\downarrow\}$ le serait aussi. ■

Corollaire 1.6.8 *Pour tout k, il existe un sous-ensemble de N^k récursivement énumérable qui n'est pas récursif, il existe sous-ensemble de $\mathbb{N}^k$ qui n'est pas récursivement énumérable.*

Démonstration. Immédiat pour $k = 1$ d'après le théorème 1.6.6, l'ensemble diagonal étant r.e. ; son complémentaire n'est donc pas r.e. (d'après la proposition 1.6.5 mais cela a été montré directement). On le déduit par codage pour k quelconque (lemme 1.6.2). ■

Le problème de l'arrêt et celui de l'arrêt diagonal constituent les deux problèmes de base pour montrer, plus tard, l'indécidabilité d'autres problèmes algorithmique.

Soit $A \subseteq \mathbb{N}^k$ et $f : A \longrightarrow \mathbb{N}$ une fonction partielle. On dit que g est une extension totale de f si g est une fonction totale et, pour tout $\overline{x} \in A$, $f(\overline{x}) = g(\overline{x})$. Le résultat suivant est une conséquence assez directe de l'indécidabilité du théorème de l'arrêt. On en donne néanmoins une preuve indépendante jouant encore une fois avec la méthode de diagonalisation.

Proposition 1.6.9 *Il existe une fonction partielle récursive $f : \mathbb{N} \longrightarrow \mathbb{N}$ qui n'a pas d'extension totale récursive.*

Démonstration. Soit φ^1 la fonction universelle à une variable. On définit une fonction récursive f qui diffère en tout point de la diagonale de φ^1, par exemple :

$$f(x) = 1 \dot{-} \varphi^1(x,x).$$

Supposons qu'il existe une fonction totale g récursive qui étend f. Dans ce cas, il existe un indice m tel que $g(x) = \varphi^1(m,x)$. En évaluant g sur l'argument m on voit que $f(m) = 1 \dot{-} g(m)$ et donc f est définie en m et $f(m) \neq g(m)$. Donc g ne peut être une extension récursive de f car les deux fonctions ne coincident pas en m. ■

1.6.4 Ensembles séparables, théorème de Rice

Soient $A \subseteq \mathbb{N}^k$ et $B \subseteq \mathbb{N}^k$ deux ensembles disjoints. On dit qu'un ensemble $C \subseteq \mathbb{N}^k$ *sépare* A et B si $A \subseteq C$ et $B \subseteq \mathbb{N}^k \backslash C$; A et B sont dits *récursivement séparables* s'il existe un tel ensemble C qui est récursif.

Théorème 1.6.10 *Il existe deux ensembles disjoints récursivement énumérables $A \subseteq \mathbb{N}^k$ et $B \subseteq \mathbb{N}^k$ qui ne sont par récursivement séparables.*

Démonstration. Dans la preuve de la proposition 1.6.9, la fonction (partielle) f considérée a pour image l'ensemble $\{0,1\}$. Considérons les ensembles $A = \{x \mid f(x) = 1\}$ et $B = \{x \mid f(x) = 0\}$ (A et B ne forment pas une partition de $\mathbb{N}$ car f peut ne pas être défini pour certains x). A et B sont clairement récursivement énumérables. Supposons qu'il existe un ensemble récursif C tel que $A \subseteq C$ et $B \subseteq \mathbb{N}^k \backslash C$. Dans ce cas, la fonction caractéristique de C, qui est totale et récursive étend f en contradiction avec la Proposition 1.6.9 ■.

Le résultat qui va suivre est très riche de conséquences. Il montre, en quelque sorte, que toute propriété des machines qui ne dépend que des entrées-sorties, toute propriété d'un ensemble de programmes qui ne dépend que de ce que calcule la machine (pas de la façon dont elle le calcule), est soit triviale soit indécidable.

Théorème 1.6.11 (Théorème de Rice) *Soit F un ensemble de fonctions (partielles) récursives à k variables tel que $F \neq \emptyset$ et tel qu'il existe une fonction récursive partielle qui n'est pas dans F. Alors l'ensemble $I_F = \{m \in \mathbb{N} \mid \varphi_m^k \in F\}$ n'est pas récursif.*

On peut réénoncer le théorème de Rice ainsi :

Autre énoncé du théorème de Rice. *Soit $k \in \mathbb{N}^*$ et $E \subset N$ tels que :*

i. $\left(i \in E \text{ et } \varphi_i^k = \varphi_j^k\right) \Rightarrow j \in E$,

ii. $E \neq \emptyset$ et $E \neq \mathbb{N}$,

alors E n'est pas récursif.

On dit d'un ensemble ou d'une prédicat qui satisfait la condition i qu'elle est *extensionnelle* (pour les fonctions récursives partielles d'arité k), ce qui traduit le fait que le prédicat sur les entiers ne dépend que de la fonction partielle (au sens ensembliste, le graphe de celle-ci, donc l'extension) dont l'indice est l'entier, et non de l'entier lui-même (ni donc de la machine dont il est le code). Le théorème de Rice exprime donc qu'une propriété extensionnelle non triviale est indécidable.

Le problème de l'arrêt fournit typiquement un exemple de propriété extensionnelle. Le théorème de Rice se démontre en se ramenant à celui-ci.

Démonstration. On appelle v la fonction nulle part définie. On suppose que $v \in F$ et soit $g \in \mathscr{F} \setminus F$. Soit H un ensemble récursivement énumérable non récursif (comme l'ensemble DIAG). On définit la fonction $f : \mathbb{N}^{k+1} \to \mathbb{N}$ suivante :

$$f(n, \overline{x}) = \begin{cases} g(\overline{x}) \text{ si } n \in H \\ \text{non définie si } n \notin H. \end{cases}$$

La fonction f est récursive (car définie par cas à partir d'un prédicat récursivement énumérable H et d'une fonction récursive g [7]). On remarque que, pour n fixé, la fonction f est soit égale à v soit égale à g. On va exploiter ce fait. Notons q un indice de la fonction f. Les deux situations suivantes sont possibles :

- Soit $n \in H$. Dans ce cas, $\varphi^{k+1}(q, n, \overline{x}) \downarrow$ et $\varphi^{k+1}(q, n, \overline{x}) = g(\overline{x})$ pour tout $\overline{x}$. Par le Théorème s_n^m, il suit alors que $\varphi^k(s_1^{k+1}(q,n), \overline{x}) =_{\text{def}} \varphi^k_{s_1^{k+1}(q,n)}(\overline{x}) = g(\overline{x})$ et donc $s_1^{k+1}(q,n) \notin I_F$.
- Soit $n \notin H$. Dans ce cas, $\varphi^{k+1}(q, n, \overline{x})$ n'est défini pour aucun $\overline{x}$ ce qui la rend égale à la fonction v et donc $s_1^{k+1}(q,n) \in I_F$.

On a donc l'équivalence :

$$n \in H \iff s_1^{k+1}(q,n) \notin I_F$$

Supposons que l'ensemble I_F soit récursif. Dans ce cas, on peut déterminer l'appartenance de n à H indirectement en calculant $s_1^{k+1}(q,n)$ (cette fonction est récursive totale) et en testant son appartenance à I_F. Ceci contredit le fait que H n'est pas récursif. ■

Examinons maintenant quelques conséquences du Théorème de Rice

Corollaire 1.6.12 *Les problèmes suivants sont indécidables.*

- *Étant donnée une machine M, déterminer si la fonction calculée par M est une certaine fonction f fixée (i.e. l'ensemble des codes d'une fonction récursive f n'est pas récursif).*
- *Étant donnée une machine M, déterminer si l'ensemble des mots acceptés par M (i.e. le langage accepté par M) est fini.*
- *Étant données deux machines M et N, déterminer si elles calculent la même fonction.*

7. Prouvez le formellement ou donner le principe de l'algorithme calculant f

Exercice 16 Soient $F \subseteq \mathcal{F}$, $G \subseteq \mathcal{F}$ deux ensembles de fonctions récursives à k variables telles que $dom(F) \cap Dom(G) = \emptyset$. Montrer que les ensembles $\{m : \varphi_m^k \in F\}$ et $\{m : \varphi_m^k \in G\}$ sont récursivement inséparables.

1.7 Réductions

Les preuves du Théorème de Rice et du Théorème 1.6.7 sont d'une nature particulière : elles sont obtenues par *réduction* à un autre problème. On peut formaliser la notion de réduction de plusieurs manières, plus ou moins libérales suivant les moyens mis en jeu.

1.7.1 Réduction "many-one"

Définition 1.7.1 (réduction many-one) Soient $A \subseteq \mathbb{N}^k$ et $B \subseteq \mathbb{N}^l$ deux prédicats. On dit que A se réduit à B par *réduction many-one*, noté $A \leq_m B$, s'il existe une fonction [8] $f : \mathbb{N}^k \to \mathbb{N}^l$ récursive totale telle que, pour tout $\overline{x} \in \mathbb{N}^k$:

$$A(\overline{x}) \iff B(f(\overline{x})).$$

Clairement, la réduction *transmet* la difficulté : si A n'est pas récursif et que A se réduit à B par réduction many one, alors B n'est pas récursif. Sinon, un algorithme évident pour énumérer A serait obtenu, sur toute entrée n, en calculant $f(n)$ et en testant si $f(n) \in B$. Cet argument montre aussi que l'on peut utiliser *positivement* la réduction pour décider A si on sait décider B. Pour résumer :

Lemme 1.7.2 *Soient A, B des prédicats tels que $A \leq_m B$.*
- *Si B est récursivement énumérable, alors A est récursivement énumérable.*
- *Si B est récursif, alors A est récursif*
- *Si de plus C est un prédicat tel que $B \leq_m C$, alors $A \leq_m C$. La réduction est transitive.*

On note $A \equiv_m B$ ssi $A \leq_m B$ et $B \leq_m A$.

Une dernière caractéristique fondamentale des réductions est qu'elles permettent parfois de montrer que certains prédicats sont représentatifs des problèmes les plus difficiles d'une classe autrement dit qu'ils sont *complets*.

Définition 1.7.3 Un prédicat r.e. A est m-complet si, pour tout prédicat r.e. B, $B \leq_m A$.

On peut montrer le résultat suivant.

Proposition 1.7.4 *Un prédicat A est récursivement énumérable ssi $A \leq_m$* DIAG.

Démonstration. Si $A \leq_m$ DIAG alors A est récursivement énumérable par le lemme précédent car DIAG est r.e..

La preuve de réduction vers le problème ARRET serait plus immédiate. Pour DIAG un petit détour est nécéssaire. Soit A un prédicat récursivement énumérable domaine d'une fonction partielle récursive $f : \mathbb{N}^k \to N$. Soit h la fonction récursive partielle définie par :

$$h(\overline{x}, y) = \begin{cases} 1 \text{ si } A(\overline{x}) \\ \text{non définie sinon} \end{cases}$$

Soit m un indice de h. Par le théorème s-m-n, on a :

$$\varphi(m, \overline{x}, y) = \varphi(s_2^1(m, \overline{x}), y) \text{ et } \overline{x} \in A \iff \varphi(s_2^1(m, \overline{x}), s_2^1(m, \overline{x})) \downarrow \iff s_2^1(m, \overline{x}) \in \text{DIAG} \qquad \blacksquare$$

8. On peut voir une fonction $f : \mathbb{N}^k \to \mathbb{N}^l$ comme une collection de fonctions $f_1, ..., f_l : \mathbb{N}^k \to \mathbb{N}$

1.7.2 Réduction de Turing

Définition 1.7.5 Soit g une fonction totale (resp. partielle), l'ensemble des fonctions récursives totales (resp. partielles) en g est le plus petit sous-ensemble de fonctions :

1. contenant la fonction nulle, la fonction successeur et les projections ainsi que la fonction g
2. clos par schéma de composition, récursion primitive et schéma de minimisation appliqué aux fonctions régulières (resp. schéma de minimisation).

Un prédicat A sera récursif (partiel ou total) en B si sa fonction caractéristique est récursive (partielle ou totale) en la fonction caractéristique de B.

La définition précédente exprime le fait que f est calculable modulo "un oracle" [9] pour la fonction g. Elle n'est effectivement récursive que si g l'est.

La plupart des résultats sur les fonctions récursives se généralisent au cas des fonctions récursives relativement à une fonction ou un prédicat (forme normale de Kleene, notion d'indice de machine etc...). On donne ci-dessous sans preuve une version possible de la forme normale de Kleene dans le cas restreint (mais qui est celui qui nous intéresse dans le cadre des réductions et suffisante pour les résultats à montrer) d'une fonction récursive partielle en une (seule) fonction totale. La preuve découle facilement de celle de la proposition 1.5.11. Soit $g : \mathbb{N}^k \to \mathbb{N}$ une fonction récursive totale et $y \in \mathbb{N}$. On rappelle que $\tilde{g}(y)$ est définie par :

$$\tilde{g}(y) = [g(\overline{0}); ...; g(\overline{y})]$$

Proposition 1.7.6 *Pour chaque entier n, il existe un prédicat récursif primitif $T^n \subseteq \mathbb{N}^n$ et une fonction primitive récursive $U : \mathbb{N} \to \mathbb{N}$ tels que pour toute fonction $f : \mathbb{N}^n \to N$ récursive partielle en une fonction totale g, il existe un entier m tel que :*

1. $\exists s\ T^n(m, x_1, \ldots, x_n, \tilde{g}(s), s) \Leftrightarrow f(x_1, \ldots, x_n) \downarrow$
2. $T^n(m, x_1, \ldots, x_n, \tilde{g}(s), s) \Rightarrow U(s) = f(x_1, \ldots, x_n)$
3. $\big(T^n(m, x_1, \ldots, x_n, \tilde{g}(s), s) \text{ et } T^n(m, x_1, \ldots, x_n, \tilde{g}(s'), s')\big) \Rightarrow s = s'$

Le fait que g est totale nous permet d'utiliser une représentation finie de celle-ci (en fait, une suite des valeurs jusqu'à une certaine étape). On pose la définition suivante à l'image de ce que l'on a fait dans le cadre des fonctions récursives.

Définition 1.7.7 Soit $A \subseteq N^k$ On note φ_i^A la fonction en n variables d'indice i récursive partielle en A i.e.

$$\varphi_i^A(\overline{x}) \equiv U(\mu s.[T^n(i, \overline{x}, \tilde{g}(s), s)]).$$

De même, on peut parler de l'approximation de φ_i^A à l'étape t [10].

$$\varphi_i^A(\overline{x})[t] = \begin{cases} \varphi_i^A(\overline{x}) \text{ si } \exists s < t\ T^n(i, \overline{x}, \tilde{g}(s), s)) \\ \text{non définie sinon} \end{cases}$$

On définit maintenant la réduction dite de Turing.

Définition 1.7.8 Un prédicat A est Turing réductible à un prédicat B si A est récursif (total) en B i.e. si la fonction caractéristique de A est récursive totale en celle de B.

On considère une réduction comme une fonction totale ici. Bien entendu, comme dans le cas classique, la notion d'indice de programme concerne les fonctions partielles.

Noter que si $A \leq_m B$ alors $A \leq_T B$: les réductions de Turing sont plus générales que les réductions many-one. La proposition qui suit est donc un corollaire immédiat de celle qui a été donnée plus haut.

Proposition 1.7.9 (réduction de Turing) *Si un prédicat A est récursivement énumérable alors $A \leq_T$* DIAG.

9. on invite le lecteur à réfléchir à la façon d'introduire une notion de calcul avec oracle pour les machines de Turing ou les machines à registres

10. le terme étape est un peu trompeur car t n'est pas le temps de calcul (qui correspondrait plutôt à la longueur de la suite s

Par contre, la réciproque n'est plus vraie car $\overline{\text{DIAG}} \leq_T$ DIAG où $\overline{\text{DIAG}} = \mathbb{N} \backslash$ DIAG.

Définition 1.7.10 Un prédicat r.e. A est Turing-complet si, pour tout prédicat r.e. B, $B \leq_T A$.

De même que pour les réductions many-one, on notera $A \equiv_T B$ ssi $A \leq_T B$ et $B \leq_T A$. Les relations $\leq_m$ tout comme $\leq_T$ sont des relations d'équivalences. En général, on appelle *degré* une classe d'équivalence pour $\leq_T$. Une question tout à fait naturelle et importante, formulée par Post est la suivante :

Problème de Post : Existe-t'il des ensembles récursivement énumérables qui ne sont ni récursifs, ni Turing complets ?

Dans la suite, on va s'attacher à donner une preuve de la réponse positive à cette question (difficile).

1.8 Une méthode des priorités

La solution au problème de Post se trouve dans le résultat suivant qui a nécessité beaucoup d'efforts et l'introduction d'une nouvelle approche : la méthode des priorités. On présentera ici une variante de cette dernière appelée *finite injury priority method.*

Théorème 1.8.1 (Friedberg (1957), Müchnik (1956)) *Il existe des prédicats récursivement énumérables A qui ne sont ni récursifs, ni Turing-complet.*

Jouer avec les priorités. On suppose que l'on se donne un ensemble de conditions P_i, N_i pour $i \in \mathbb{N}$ à réaliser au cours du temps. Supposons que, pour tout $i \in N$:

1. Si P_i devient vrai à une étape t elle reste vrai indéfiniment
2. Au moment où P_i devient vrai, elle peut éventuellement falsifier une condition N_j avec $j \geq i$
3. Lorsque N_i devient vraie, elle ne peut falsifier aucune autre condition N_j ($\neq N_i$) ou P_j
4. Pour toute étape t, il existe $s \geq t$ tel que, si P_i (resp. N_i) est fausse en t alors P_i (resp. N_i) devient vrai en s

Alors :

Fait 1.8.2 *Sous les conditions ci-dessus, pour tout $i \in \mathbb{N}$, il existe $t_i \in \mathbb{N}$ tels que, pour tout $t \geq t_i$, P_i et N_i sont vrais en t.*

Cette propriété vient du fait qu'une propriété N_i ne peut être invalidée qu'un nombre fini de fois : au moment où l'un des P_j d'indice inférieur ($j \leq i$) devient vrai. Comme aucun P_j ne peut être invalidé, il y a au plus $i+1$ invalidations possibles pour N_i. Une fois passées, à partir du moment où N_i redevient vrai (cf. dernière condition), elle le reste.

Démonstration. On va construire en parallèle deux ensembles A et B, tous les deux récursivement énumérables (r.e.) tels que le complémentaire de A est infini et pour tout indice i de machine :

$\mathbf{P_i}$ Si W_i est infini alors $W_i \cap A \neq \emptyset$

$\mathbf{N_i}$ $B \neq \varphi_i^A$

où W_i est le domaine de la i^{eme} fonction récursive partielle.

Si ces conditions sont satisfaites, alors A n'est pas récursif car sinon son complémentaire serait un ensemble récursivement énumérable donc un W_i infini pour un certain indice i. Or la première condition $W_i \cap A \neq \emptyset$ impliquerait $A \cap N \backslash A \neq \emptyset$ ce qui est absurde. La seconde condition assure le fait que A n'est pas Turing-complet. Pour obtenir la condition N_i, il suffit d'exhiber une entrée x_i, un *témoin*, telle que $B(x_i) \neq \varphi_i^A(x_i)$.

La construction des ensembles A et B se fait incrémentalement. Dans la suite, pour tout entier t, on notera A^t et B^t l'état des ensembles A et B à l'étape t. La propriété suivante sera vérifiée :

$$\forall t \in \mathbb{N}, A^t \subseteq A^{t+1} \text{ et } B^t \subseteq B^{t+1}$$

On démarre avec $A^0 = B^0 = \emptyset$. Quelques notations :

– On rappelle que $\varphi_i^A(x)[t]$ est l'approximation de la machine avec oracle A d'indice i sur l'entrée $\overline{x}$ à l'étape t.
– On pose $A^{t,v} = A^t \cap \{0,...,v\}$.

Pour une entrée donnée x, un indice de machine i et un nombre d'étape t, on définit :

$$u(i,t,x) = \begin{cases} \mu v.[\varphi_i^{A^{t,v}}(x)[t]\downarrow] \\ 0 \text{ sinon} \end{cases}$$

Si le calcul de la machine i sur l'entrée x s'arrête au bout de t étapes en utilisant A^t comme oracle, alors la fonction u renvoie, en quelque sorte, la taille du plus petit segment initial de A^t qu'il est nécessaire d'utiliser pour que le calcul s'arrête.

Une difficulté est de satisfaire la condition N_i car, pour chaque i, il faut trouver un témoin x_i adéquat. Une autre est qu'à chaque fois que l'ensemble A est mis à jour en ajoutant un élément x (pour satisfaire une condition P_i) par exemple à l'étape $t+1$, les calculs de certains $\varphi_j^{A^t}(x_j)$ peuvent être remis en question : si, par exemple, $v = r(j,t,x_j)$ est plus grand que x cela indique que le calcul de $\varphi_j^{A^t}(x_j)$ a pu utiliser le fait que $x \not\in A^t$ et avec la nouvelle information contradictoire ($x \in A^{t+1}$), la conclusion de ce calcul pourrait ne plus être la même. La condition N_j ne serait alors plus forcément satisfaite si elle l'était auparavant... Il faudrait donc recommencer pour N_j et chercher un nouveau témoin. L'idée pour éviter une remise en cause perpétuelle des conditions est d'utiliser une liste de priorité entre celles-ci : on ne met à jour A pour satisfaire une condition P_i que si les seules conditions invalidées par cette mise à jour sont de priorités inférieures. Etablissons par exemple la liste de priorité :

$$P_0 < N_0 < P_1 < \ldots < N_i < P_{i+1} < N_{i+1} < \ldots$$

On accepterait la mise à jour de A pour satisfaire P_i seulement dans l'hypothèse où les conditions invalidées par cette mise à jour sont parmi :

$$N_i, P_{i+1}, N_{i+1}, \ldots.$$

On détaille maintenant la stratégie à adopter pour satisfaire l'ensemble des conditions. On veut notamment trouver un ensemble (infini) de témoins $x_0, x_1, ..., x_i, ...$ qui permettent à chaque condition N_i d'être satisfaite. A chaque étape t, on dispose d'un ensemble de témoins candidats x_i^t, $i \leq t$ (on ne peut pas prendre en considération toutes les conditions à chaque étape) et pour chaque indice, de la valeur de $u(i,t,x_i^t)$. On supposera que l'on cherche à satisfaire les conditions dans l'ordre des priorités (même si formellement, cela ne change pas grand chose...) : d'abord P_0 puis N_0, P_1, etc.

Stratégie pour satisfaire P_i. A chaque étape $t+1$, pour tout $i \leq t$, si $W_i \cap A^t = \emptyset$ et si

$$\exists x\ x \in W_i^t \wedge x > \max(A^t) + 1 \wedge (\forall j \leq i)(x > u(j,t,x_j^t))$$

alors prendre le plus petit tel x et mettre à jour A : $A^{t+1} := A^t \cup \{x\}$. Après cette étape, soit il n'y a pas eu de mise à jour et dans ce cas aucune condition n'est invalidée (et P_i n'est pas encore satisfaite). Soit il y en a eu une alors, dans ce cas, la condition P_i est satisfaite et, du fait de la contrainte $(\forall j \leq i)(x > u(j,t,x_j^t))$ sur x, aucune condition d'indice inférieur (prioritaire sur i) n'est invalidée. Seules des conditions N_j avec $j \geq i$ (donc moins prioritaires) ont pu l'être. On dira dans ce cas que N_j est invalidée en $t+1$.

Stratégie pour satisfaire N_i. A chaque étape $t+1$, pour chaque $i \leq t$, on exécute $\varphi_i^{A^t}$ sur l'entrée x_i^t jusqu'à l'étape t. Si N_i n'est pas invalidée en $t+1$ (par un $x \in A^{t+1} \setminus A^t$ plus petit que $u(j,t,x_i^t)$) et

$$\text{si } \varphi_i^{A^t}(x_i^t)[t] = 0 \text{ et } x_i^t \not\in B^t \text{ alors } B^{t+1} := B^t \cup \{x_i^t\}$$

sinon, ne rien faire. Si une mise à jour de B a lieu, on a bien (peut-être provisoirement) un témoin x_i^t satisfaisant N_i en $t+1$: $B^{t+1}(x_i^t)$ et $\varphi_i^{A^t}(x_i^t)[t] = 0$. Noter que cette condition est la seule possibilité de mettre à jour B par x_i^t. Si $x_i^t \not\in B^t$ et $\varphi_i^{A^t}(x_i^t)[t] = 1$ ou $\varphi_i^{A^t}(x_i^t)[t]$, alors x_i^t est toujours un témoin à considérer.

Remarque : la condition réelle d'invalidité pour une condition N_i en $t+1$ correspond à la situation où $x_i^t \in B^t$ avec, soit $\varphi_i^{A^{t+1}}(x_i^t)[t+1] = 1$, soit $\varphi_i^{A^{t+1}}(x_i^t)[t+1]\uparrow$ du fait de la mise à jour de A^t en A^{t+1}.

Dans tous les cas, à l'étape suivante, nous avons une nouvelle liste de témoins potentiels. Si N_i n'a pas été invalidée ou si $x_i^t \notin B^{t+1}$, on pose $x_i^{t+1} = x_i^t$. Sinon, on prend pour x_i^{t+1} une nouvelle valeur non considérée jusque là (par exemple, plus grande que celle des autres témoins considérés jusque là [11]).

Que peut-on dire de toutes les contraintes à satisfaire ? Clairement, A et B sont récursivement énumérables : on a un algorithme pour les construire de façon incrémentale. Et, contrairement aux conditions N_i et P_i, on ne revient à aucun moment sur le choix des éléments que l'on met dans A et B. On voit que le complémentaire de A est infini (cf. condition $\max(A^t)+1$ qui rejette d'office hors de A au moins un élément à chaque nouvelle mise à jour de A). Il reste à montrer que toutes les conditions N_i et P_i vont être satisfaites.

Pour tout $i \in \mathbb{N}$, la condition N_i est satisfaite. En effet, celle-ci ne peut être invalidée qu'un nombre fini de fois : au moment précis où une condition P_j est satisfaite, avec $j < i$ à une certaine étape $t+1$, et seulement pour un $x \in A^{t+1} \setminus A^t$ tel que $x < u(i,t,x_i^t)$. Donc, il existe un instant t_i, tel que pour tout $t \geq t_i$, N_i ne peut plus être invalidée en t. Posons $x_i = x_i^{t_i}$.

Soit $\varphi_i^A(x_i)\uparrow$ dans ce cas, comme la fonction φ_i^A obtenue n'est pas définie en x_i, donc pas totale, on a bien $B \neq \varphi_i^A$. Soit il existe une étape $t \geq t_i$ telle que $\varphi_i^{A^t}(x_i)[t]\downarrow$. Considérons le premier tel t. Dans ce cas, comme la condition N_i ne peut plus être invalidée alors :

$$\varphi_i^{A^t}(x_i) = \varphi_i^A(x_i) \text{ i.e. } \varphi_i^{A^t}(x_i) = \varphi_i^{A^s}(x_i), \forall s \geq t$$

Or, pas construction, on va avoir, $B(x_i) \neq \varphi_i^A(x_i)$. En effet :

- soit $\varphi_i^{A^t}(x_i) = 0$ et dans ce cas $x_i \in B^{t+1} \subseteq B$,

- soit $\varphi_i^{A^t}(x_i) = 1$ (et $\varphi_i^{A^s}(x_i)[s]\uparrow$ pour $t_i \leq s \leq t$) et, dans ce cas $x_i \notin B^{t+1}$ (on aura toujours évité de mettre x_i dans B).

Enfin, pour chaque $i \in \mathbb{N}$ la condition P_i est satisfaite. Si $r_i \geq \max_{j\leq i} u(j,t_j,x_j)$ et $t = \max_{j \leq i} t_j$ alors comme W_i est infini, il existe forcément x tel que :

$$W_i(x) \wedge x \geq \max_{j \leq i} x_j \wedge (x > r_i).$$

Il suffit d'attendre l'étape t ou une étape ultérieure quand la condition ci-dessus sera satisfaite.

■

11. On peut imaginer un autre mode de sélection des témoins

Chapitre 2

Modèles de calcul : compléments

Dans ce chapitre, on présente brièvement deux autres modèles de calcul : les RAM (Random Access Machine) et les machines de Turing. Le premier est obtenu par une extension des opérations permises sur une machine à registres. Bien évidemment équivalente à cette dernière, elle va nous permettre de tendre vers un modèle dans lequel l'écriture des algorithmes est naturelle. Le second modèle (la machine de Turing) est historiquement important. Il est par essence plus rudimentaire et l'écriture d'un algorithme complexe dans ce modèle est toujours fastidieuse mais il nous servira comme point de départ d'un certain nombre de résultats dont les preuves sont plus faciles à écrire en partant de modèles de calcul plus "atomiques".

2.1 Machines à accès indirect

Le modèle des machines à registres considéré jusque là utilise un nombre de registres finis qui, évidemment, peuvent contenir des entiers arbitrairement gros. Il peut être plus pratique de supposer que l'on a à disposition un nombre potentiellement infini de registres. Bien sûr, dans ce cas, il est nécessaire d'avoir des instructions permettant d'atteindre tout registre. On va voir que rajouter cette possibilité ne change rien au pouvoir de calcul du modèle. Plus précisement, on ajoute à notre modèle la possibilité de considérer la valeur d'un registre comme étant l'indice d'un registre auquel on va pouvoir accéder librement. Par soucis de réalisme, en plus des registres classiques $R_0, R_1, \ldots.$, cette fois en nombre a priori non limité, on introduit un registre spécial A, appelé accumulateur et on étend le jeu d'instruction d'un programme structuré par les deux instructions suivantes.

1. le registre A reçoit la valeur du registre dont l'adresse (l'indice) se trouve dans le registre R_j :

$$A := \langle R_j \rangle,$$

2. le registre dont l'adresse se trouve dans A reçoit la valeur du registre R_j :

$$\langle A \rangle := R_j.$$

Par exemple, si $A = 2$, $R_3 = 100$ et $R_{100} = 0$ alors, l'instruction $A := \langle R_3 \rangle$ donne 0 pour nouvelle valeur à A.

Ces instructions sont dites d'*adressage indirect*. Les machines à registres (avec programme goto ou programme structuré) utilisant un nombre potentiellement infini de registres et ses instructions complémentaires sont appelées machines *RAM* (pour Random Access Machine).

On considère qu'au début du calcul d'une machine RAM, tous les registres qui ne contiennent pas les entrées sont initialisés à 0.

Proposition 2.1.1 *Soit une fonction partielle $f : \mathbb{N}^k \to \mathbb{N}$ calculable par une machine RAM alors elle est calculable par une machine à registres usuelle.*

Démonstration. (Principe général). Soit M une machine RAM calculant une fonction f. On supposera que les entrées $n_1,\ldots,n_k$ de f sont initialement dans les registres $R_1,\ldots,R_k$. Principalement, il suffit de stocker la suite des valeurs de registres utilisés par la machine simulée dans un seul registre de la machine qui simule. Appelons T le registre qui contiendra l'ensemble de ses informations. Il faut noter qu'à chaque étape du calcul seul un nombre fini de registres a pu être utilisé. A chaque instant, l'entier contenu dans T est le code d'une liste de la forme :

$$l = [c_1; c_2; \ldots; c_n]$$

où chaque $c \in l$ est de la forme $[i; r_i]$ où r_i sera le contenu du registre R_i. Tous les numéros (adresses) de registres visités ainsi que leur contenu seront ainsi stockés petit à petit dans T.

On a déjà construit les fonctions qui permettent de tester l'appartenance d'un élément à une liste (mem(x,l)), de concaténer deux listes ($l @ l'$), etc. On laisse au lecteur le soin de construire :

- la fonction memreg définie par memreg(i,l) $= c$ pour $c \in l$ tel que $\pi_1^2(c) = i$. Si aucun élément $c \in l$ ne vérifie $\pi_1^2(c) = i$, on pose memreg(i,l) $= 0$. On posera val(i,l) $= \pi_1^2$(memreg(i,l)).
- la fonction replace définie par replace(i,x,l) est la liste l dans laquelle l'élément c tel que $\pi_1^2(c) = i$ est remplacée par $[i,x]$. I.e. la valeur de la seconde coordonnée de c est remplacée par x.

Dans une première étape, facultative, on peut initialiser le registre T par :

$$T = [[0;0];[1;n_1];\cdots;[k,n_k]]$$

de façon à ce que T contienne l'ensemble des valeurs d'entrées (et $R_0 = 0$). Dans le reste de la simulation, on utilisera juste les registres A et T (ainsi que les registres nécessaire pour calculer les fonctions ci-dessus).

L'instruction $A := \langle R_j \rangle$ est remplacé par les étapes suivantes [1] :

```
si memreg(j,T) = 0                         (* si le registre R_j n'a pas déjà été visité *)
    alors A := 0                                              (* A prend la valeur 0 *)
    sinon si memreg(val(j,T),T) = 0        (* si le registre ⟨R_j⟩ n'a pas déjà été visité *)
              alors A := 0                                    (* A prend la valeur 0 *)
              sinon A := val(val(j,T),T)   (* A prend la valeur du registre d'indice R_j *)
```

L'instruction $\langle A \rangle := R_j$ est remplacé par les étapes suivantes :

```
si memreg(A,T) = 0                      (* si le registre d'indice A n'a pas déjà été visité *)
    alors si memreg(j,T) = 0                  (* le registre ⟨A⟩ devient référencé dans T *)
              alors T := [A,0] @ T                               (* ... avec la valeur 0 *)
              sinon T := [A,val(j,T)] @ T                      (* ... avec la valeur de R_j *)
    sinon si memreg(j,T) = 0
              alors T := replace(A,0,T)               (* le registre ⟨A⟩ prend la valeur 0 *)
              sinon T := replace(A,val(j,T),T)  (* le registre ⟨A⟩ prend la valeur de R_j *)
```

Il faut aussi remplacer l'ensemble des instructions classiques $R_i := R_i + 1$, $R_i := R_i - 1$, $R_i := 0$, $R_i := R_j$ ainsi que les tests sur la valeur d'un registre par des manipulations directes sur T. On laisse les détails au lecteur. ■

L'avantage de ce nouveau modèle est que cette fois on peut envisager de borner la taille des registres (par une fonction dépendant de l'entrée par exemple) et éviter ainsi une situation peu naturelle où la taille des registres peut gonfler arbitrairement. Cela correspond mieux aussi à l'architecture d'un processeur avec une mémoire de taille a priori non bornée mais segmentée (les registres de ce modèle) et des "vrais" registres en nombre limité sur lesquelles sont réalisés les opérations arithmétiques et ayant accès à la mémoire pour y chercher les données à manipuler (ce rôle est joué ici par l'accumulateur).

On a choisi ici un modèle un peu hybride. On se convaincra aisément que :

1. cet "algorithme" et le suivant peuvent facilement s'écrire dans la syntaxe des RAMs une fois que l'on a écrit les programmes pour les fonctions val, replace, etc.

– on peut augmenter le nombre d'accumulateur voir généraliser l'adressage indirect en permettant des instructions du type $\langle R_i \rangle := R_j$ pour toute paire de registres..
– on peut s'autoriser un nombre potentiellement non borné de registres en entrée. Par exemple, une liste finie peut être codée comme un nombre représentant la suite des éléments et être stockée dans un seul registre d'entrée R_1. Elle peut aussi être stockée élément par élément dans une suite de registres, ceux-ci étant accessible par manipulation du registre d'accumulation.

Avec l'adressage indirect, on récupère de façon naturelle la notion de pointeur, de tableau et donc de structure de données évoluée (sans avoir à passer par un codage arithmétique). Le modèle obtenu correspond à un pseudo-langage de programmation rudimentaire.

2.2 Machines de Turing

On présente brièvement ci-dessous une autre notion de machine due à Turing. Le principe est celui d'un programme (une suite finie d'instructions élémentaires) agissant sur un espace de travail par modifications locales.

2.2.1 Machines de Turing déterministes

Une machine de Turing M est un modèle de calcul simple que l'on peut décrire informellement de la façon suivante. Elle dispose d'un ensemble fini d'états Q, d'un ruban (que l'on supposera infini vers la droite mais borné vers la gauche) composé de cellules (ou cases) pouvant contenir chacun un symbole d'un alphabet Γ. Elle dispose aussi d'une tête de lecture qui se trouve à chaque étape au niveau d'une case du ruban.

Le calcul se déroule comme suit. A chaque instant t du calcul, la machine est dans un état particulier $q \in Q$, lit le symbole γ pointé par sa tête de lecture et... se retrouve à l'instant $t+1$ dans un nouvel état $q' \in Q$, remplace γ par un nouveau symbole $\gamma' \in \Gamma$ et soit bouge sa tête de lecture à droite ou à gauche soit la laisse immobile. La transformation à appliquer (qui dépend de γ et q) est déterminée par le *programme* de M (que l'on appelle sa fonction de transition).

La machine débute son calcul dans un état dit *initial* avec un mot d'entrée $\overline{w} = w_1 \dots, w_n \in \Sigma^n \subseteq \Gamma^n$ écrit sur les n premières cases du ruban. On appelle $|w| = n$ la taille de w. Sans ces conditions initiales, tous les calculs seraient identiques. Elle termine son calcul quand elle atteint un état final acceptant (q_a) ou rejetant (q_r). Notons qu'une machine peut ne pas terminer son calcul...

Définition 2.2.1 (machine de Turing déterministe) Une machine de Turing *déterministe* (dite "à un ruban") est la donnée d'un tuple $M = (Q, \Gamma, \Sigma, \delta, q_0, q_a, q_r)$ où :
– Q est un ensemble fini d'*états*,
– Γ est un ensemble fini de symboles de ruban contenant des symboles particuliers : blanc $\sqcup$, indiquant que la case est vide et $\sharp$, marqueur de la case la plus à gauche.
– $\Sigma \subseteq \Gamma$ est un ensemble fini décrivant l'alphabet d'entrée,
– $\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{\lhd, \rhd, -\}$ est une fonction de transition. Le symbole "$\lhd$" (resp. "$\rhd$", resp. "$-$" symbolise "à gauche" (resp. "à droite", resp. "immobile"),
– $q_0 \in Q$ est l'état initial, $q_a \in Q$ est l'état final acceptant et $q_r \in Q$ est l'état final rejetant.

On peut envisager plein de variantes analogues : rubans infinis des deux cotés [2], pas de distinction entre Σ et Γ, restriction à $\Sigma = \Gamma = \{0, 1, \sqcup\}$, etc.

Les machines décrites permettent d'accepter ou de refuser des mots. On peut les modifier de façon à leur faire produire une sortie et définir la notion de fonction calculable.

Définition 2.2.2 (fonction et prédicat Turing-calculable) Soit $D \subseteq \Sigma^*$, soit $f : D \to \Sigma^*$ et M une machine de Turing déterministe. On dit que M calcule la fonction f (f sera dite *Turing-calculable*) si, pour toute entrée $\overline{w} \in D$, la machine s'arrête dans l'état acceptant (après un certain nombre d'étapes) avec

2. Cela correspond d'ailleurs à la définition originale. On peut se rendre compte que les deux modèles sont équivalents. Toutefois, dans l'approche que l'on a choisi, il faut simplement supposer que la machine ne peut pas bouger sa tête de lecture vers la gauche quand elle est dans la première case où, en d'autres termes, que la fonction de transition ne contient aucune instruction du genre $\delta(q, \sharp) = (q', \sharp, \lhd)$ pour tout états q et q'

le mot $f(w)$ écrit sur son ruban. Si la fonction n'est pas définie en w (i.e. en dehors du domaine de définition D de f), la machine ne s'arrête pas.

Un prédicat A est *décidable* si sa fonction caractéristique est calculable.

On se rappelera qu'une fonction caractéristique est totale. Pour une machine de Turing donnée M, le langage accepté par M est l'ensemble des mots sur lequel M s'arrête (dans un état acceptant). A noter aussi que l'on a choisi de ne pas faire mention de l'état rejetant. Il est clair que si un calcul finit dans un état rejetant alors on peut le faire boucler indéfiniment (en ajoutant une instruction adéquate à la fonction de transition de la machine). En conséquence, on garde néanmoins cet état pratique dans les exemples.

2.2.2 Exemples

A titre d'exemple, on donne le programme d'une machine de Turing qui calcule la fonction successeur pour un entier écrit en unaire. On a $\Sigma = \{1\}$ et $\Gamma = \{\#, \sqcup, 1\}$. La partie significative de la fonction de transition est :

$$\begin{array}{rcl} (q_0,\#) & \longrightarrow & (q_1,\#,\triangleright) \\ (q_1,1) & \longrightarrow & (q_1,1,\triangleright) \\ (q_1,\sqcup) & \longrightarrow & (q_2,1,-) \\ (q_2,1) & \longrightarrow & (q_2,1,\triangleleft) \\ (q_2,\#) & \longrightarrow & (q_a,\#,-) \end{array}$$

La tête de lecture se déplace sur la droite jusqu'à rencontrer le premier blanc qu'elle remplace par un 1. Puis elle retourne au début de bande et accepte.

Le deuxième exemple teste si un mot w est un palindrôme. On suppose $w \in \{1,2\}^*$. La première instruction se place sur la première lettre significative :

$$(q_0,\#) \longrightarrow (q_0,\#,\triangleright).$$

On lit le premier symbole. On s'en souvient en passant suivant sa valeur dans l'état q_1 ou q_2 et on transforme ce symbôle en #.

$$\begin{array}{rcl} (q_0,1) & \longrightarrow & (q_1,\#,\triangleright) \\ (q_0,2) & \longrightarrow & (q_2,\#,\triangleright) \end{array}$$

On se déplace jusqu'à la fin du mot.

$$\begin{array}{rcl} (q_i,1) & \longrightarrow & (q_i,1,\triangleright) \text{ avec } i=1,2 \\ (q_i,2) & \longrightarrow & (q_i,2,\triangleright) \\ (q_1,\sqcup) & \longrightarrow & (q_3,\sqcup,\triangleleft) \\ (q_2,\sqcup) & \longrightarrow & (q_4,\sqcup,\triangleleft) \end{array}$$

On lit la dernière lettre et on la remplace par un blanc si elle est compatible avec l'état (2 si q_4 ou 1 si q_3). Sinon, on refuse.

$$\begin{array}{rcl} (q_3,1) & \longrightarrow & (q_5,\sqcup,\triangleleft) \\ (q_4,2) & \longrightarrow & (q_5,\sqcup,\triangleleft) \\ (q_3,2) & \longrightarrow & (q_r,2,-) \\ (q_4,1) & \longrightarrow & (q_r,1,-) \end{array}$$

On revient au début de bande et on recommence. On accepte lorsqu'on lit un blanc dans l'état initial (signifiant qu'il ne reste plus que des # et des ⊔ et que toutes les symboles ont été éliminés)

$$\begin{array}{rcl} (q_5,1) & \longrightarrow & (q_5,1,\triangleleft) \\ (q_5,2) & \longrightarrow & (q_5,2,\triangleleft) \\ (q_5,\#) & \longrightarrow & (q_0,\#,\triangleright) \\ (q_0,\sqcup) & \longrightarrow & (q_a,\sqcup,-) \end{array}$$

2.2.3 Extensions du modèle

Tel quel, le modèle de machine de Turing est très rudimentaire. On peut essayer de l'enrichir un peu en le dotant de plusieurs rubans.

Définition 2.2.3 (machine de Turing déterministe à k rubans) Une machine de Turing déterministe à k rubans ($k \in \mathbb{N}^*$) est une machine de Turing $M = (Q, \Gamma, \Sigma, \delta, q_0, q_a, q_r)$ dont la fonction de transition est de la forme :

$$\delta : Q \times \Gamma^k \longrightarrow Q \times \Gamma^k \times \{\lhd, \rhd, -\}^k.$$

Autrement dit, à chaque étape, des tansformations sont effectuées qui dépendent d'un seul état courant et de k symboles lus sur k rubans distincts. On adoptera la contrainte suivante (très générale) : un des rubans contient l'entrée (il sera appelée "ruban d'entrée") et il est en lecture seule i.e. le programme ne modifie à aucun moment son contenu. Dans le cas des fonctions, on suppose aussi qu'un autre ruban est destiné à recevoir le résultat du calcul. On l'appelle ruban de sortie. Les $k-1$ rubans restants sont appelés "rubans de travail".

Intuitivement, on dira que M calcule $f(w)$ en temps t si, sur l'entrée w, M s'arrête et renvoie $f(w)$ après au plus t transitions. On peut montrer que, du point de vue des fonctions calculables, les différents modèles de machine de Turing sont équivalents (on peut simuler une machines de Turing à k rubans par une machine à un seul ruban). Du point de vue du temps de calcul, on verra que les choses sont un peu plus nuancées : la simulation a un coût.

Proposition 2.2.4 *Soit $f : \mathbb{N}^n \longrightarrow \mathbb{N}$ une fonction calculable par une machine de Turing à k rubans. Alors f est calculable par une machine de Turing à 1 ruban.*

Démonstration (idée). Soit M la machine à k rubans calculant la fonction f. L'idée consiste à voir les k bandes de M comme une seule bande divisée en $2k$ pistes au moyen d'un enrichissement de l'alphabet d'entrée. On construira donc une machine M' à une seule bande sur l'alphabet $(\Gamma \times \{o, n\})^k$. Chaque case c contiendra un “symbole” $(\gamma_1, r_1, ..., \gamma_k, r_k)$ avec $r_i \in \{o, n\}$, $i \le k$. La signification sera la suivante : γ_i représente le symbole de la case c du $i$ème ruban de la machine M et $r_i = o$ si la $i$ème tête de lecture est sur cette case ($r_i = n$ sinon). A tout instant du calcul et pour chaque “piste” $2i$, une seule case vérifie $r_i = o$ (symbolisant le fait que la $i$ème tête de lecture ne peut être que sur une case à un instant donné).

On simule M par M' de la façon suivante. On va avoir besoin d'enrichir l'ensemble des états. Supposons que M est dans l'état q_α. M' parcourt tout d'abord son ruban de gauche à droite pour repérer l'emplacement de la première tête de lecture. Elle le fait en examinant la première piste. Une fois cela fait, elle récupère le symbole de la première piste γ_1 et le stocke... dans son état q_{α,γ_1}. M' fait ensuite la même chose pour toutes les pistes i, $i \le k$. À l'issue de cette première phase, M' est dans l'état $q_{\alpha,\gamma_1,...,\gamma_k}$. M' execute alors la transition de M associée à $(q_\alpha, \gamma_1, ..., \gamma_k)$, mettons $(q_\beta, \gamma_1, m_1, ..., \gamma_k, m_k)$ avec $m_i \in \{\lhd, \rhd, -\}$, et passe dans l'état correspondant $q_{\beta,\gamma_1,m_1,...,\gamma_k,m_k}$. Il ne lui reste plus qu'à faire le travail inverse de répérage des têtes de lecture pour mettre à jour les symboles et les emplacement de tête sur chaque piste.

Une analyse rapide montre que pour simuler une étape de M, la machine M' pourra avoir besoin de balayer tout l'espace utilisé par M avant l'étape en question. En effet, les têtes de lecture peuvent être arbitrairement loin les unes des autres. Simuler t étapes de M peut donc prendre jusqu'à $O(t^2)$ étapes de M'. ■

2.3 Les RAM et les machines de Turing calculent les mêmes fonctions

La proposition 2.1.1 implique que les RAM ont le même pouvoir de calcul que l'ensemble des modèles vus précédemment. Les machines de Turing sont d'apparence plus atomiques voire rudimentaires mais à nouveau l'ensemble des fonctions calculables obtenus est le même.

Proposition 2.3.1 *Une fonction $f : \mathbb{N}^k \mapsto \mathbb{N}$ est calculable par machines à registre si et seulement si elle est calculable par machine de Turing.*

Démonstration. **(Idées)** On donne tout d'abord quelques arguments pour montrer que l'on peut simuler le calcul d'une machine à registre, (ici)avec un nombre fini de registres $R_1, ...R_n$) par une machine de Turing. Un premier ruban contiendra la liste des contenus des registres $R_1, ..., R_n$ séparés par un caractère particulier.

$$R_1 \sharp R_2 \sharp \ldots \sharp R_n.$$

Simuler une instruction $R_i := R_i + 1$ (ou $R_i := R_i - 1$) se fait en :

- parcourant le ruban des registres jusqu'au $i-1$ème séparateur $\sharp$,
- recopiant R_i sur un deuxième ruban et en appliquant l'algorithme d'incrémentation pour cette valeur
- recopiant $\sharp R_{i+1} \sharp \ldots \sharp R_n$ sur un troisième ruban
- recopiant successivement $R_i + 1$ puis $\sharp R_{i+1} \sharp \ldots \sharp R_n$ sur le premier ruban

Pour simuler l'instruction if $R_i = 0$ goto p, la machine déplace sa tête sur l'endroit du ruban où se trouve le début du registre R_i (i.e. après le $k-1$ème symbole $\sharp$), teste si le mot écrit est 0 et choisit la prochaine instruction en conséquence. Comme on peut le remarquer, une seule instruction de machine à registre est parfois simulée par un nombre non bornée à l'avance d'instruction de la machine de Turing.

Pour l'autre direction, c'est à dire la simulation d'une machine de Turing M par une machine RAM, on peut procéder de la façon suivante. On posera $Q = \{1, \ldots, |Q|\}$ (avec q_0 représenté par 1 et q_a par 2), $\Sigma = \{1, \ldots, |\Sigma|\}$, $\Gamma = \{1, \ldots, |\Gamma|\}$. On supposera que le contenu du ruban d'entrée $w_1 .... w_n \in \Sigma*$ de la machine de Turing est stocké, lettre par lettre[3], dans les registres R_1,..., R_n. Tous les autres registres contiennent la valeur 0. Enfin, on utilisera deux registres additionnel, que l'on nomme E et S pour conserver la valeurde l'état et du symbole courant ainsi que l'accumulateur A pour coder la position de la tête de lecture. Au début $A = 1$, $E = 1$ et $I = 1$.

On posera $n = |\gamma| \times |Q|$ le nombre d'éléments dans la table de transitions

$$\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{+1, -1, 0\}.$$

de la machine. Le programme de la machine RAM simulant M' est le suivant :

while $E \neq 1$ do
begin
$\mathscr{S}_1$
$\vdots$
$\mathscr{S}_n$
end

où $\mathscr{S}_i$ est une suite d'instructions simulant la $i$ème transition $\delta(q, \gamma) = (q', \gamma', m')$ de la machine M.

if $E = q$ and $S = \gamma$ then
begin
$E := q'$
$\langle A \rangle := \gamma'$
$A := A + m$
$S := \langle A \rangle$
end

■

3. ces deux modèles de calcul utilisant des mécanismes d'entrée différents, il y a une certaine "tolérance" ici. On aurait pu supposer que le contenu du ruban se retrouve stocké dans un seul registre R_1 dont on déploit le contenu lettre par lettre dans les n premiers registres comme première étape de la simulation

Chapitre 3

Problèmes de décision en logique

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

3.1 Rappels : signature, structure, formule

Définition 3.1.1 *Une signature σ est la donnée d'un ensemble de symboles de constantes $c_1, \ldots, c_n$, de symboles de relation $R_1, \ldots, R_p$ et de fonctions $f_1, \ldots, f_q$.*

Chaque symbole S a une arité arité(S) qui correspond au nombre de paramètres de celui-ci

Une fonction ou relation d'arité 1 i.e. à un argument est dite unaire (voir *monadique* dans le cas d'une relation). Les constantes peuvent se voir comme des fonctions d'arité 0.

Définition 3.1.2 *Une σ-structure $\mathscr{S}$ est la donnée de :*

$$\langle Dom(\mathscr{S}), c_1^{\mathscr{S}}, \ldots, c_n^{\mathscr{S}}, R_1^{\mathscr{S}}, \ldots, R_p^{\mathscr{S}}, f_1^{\mathscr{S}}, \ldots, f_q^{\mathscr{S}} \rangle$$

où :

- *$Dom(\mathscr{S})$ (parfois noté D ou Dom) est un ensemble appelé l'univers ou le domaine de $\mathscr{S}$*
- *Chaque $c_i^{\mathscr{S}}$, $i \leq n$, est l'interprétation de c_i sur $\mathscr{S}$ i.e. un élément de D.*
- *Chaque $R_i^{\mathscr{S}}$, $i \leq p$, est l'interprétation de R_i sur D i.e. $R_i^{\mathscr{S}} \subseteq D^{r_i}$ où arité$(R_i) = r_i$.*
- *Chaque $f_i^{\mathscr{S}}$, $i \leq q$, est l'interprétation de f_i sur D i.e. $f_i^{\mathscr{S}} \subseteq D^{r_i+1}$ où arité$(f_i) = r_i$.*

Dès que le contexte est non-ambigu, on ne distingue pas le symbole de son interprétation. Si $\mathscr{S}$ et $\mathscr{S}'$ sont deux structures de domaine D sur deux signatures disjointes σ et σ' respectivement, on appelle $(\mathscr{S}, \mathscr{S}')$ la $\sigma \cup \sigma'$-structure de domaine D résultat de l'union disjointe des (relations et fonctions des) deux structures.

On ne rappelle pas ici la notion de formule du premier-ordre ni d'interprétation d'une formule dans une structure. On appelle **FO** l'ensemble des formules du premier ordre et $\mathbf{FO}_\sigma$ l'ensemble des formules du premier ordre de signature σ. Si φ est une formule du premier ordre, on note MOD(φ) l'ensemble de ses modèles finis.

Définition 3.1.3 *soit P un ensemble de σ-structures finies (on parlera de propriété). On dit que P est définissable dans $\mathbf{FO}_\sigma$ s'il existe une formule φ de signature σ telle que* MOD$(\varphi) = P$.

3.2 Problèmes de décision logique indécidables

SAT FINI PO
Entrée : Une formule close du premier-ordre φ sur une signature σ
Question : φ a-t-elle un modèle fini ?

Étant donnée une formule φ du premier ordre et une structure $\mathscr{S}$ finie, décider si $\mathscr{S} \models \varphi$ est clairement (primitif) récursif : il suffit de considérer chaque $\forall x$ comme une grande conjonction $\bigwedge_{a \in Dom(\mathscr{S})}$, chaque $\exists x$ comme $\bigvee_{a \in Dom(\mathscr{S})}$, de remplacer chaque x par sa valeur a et de tester si la formule sans quantificateur [1] ainsi obtenue est vraie dans $\mathscr{S}$. Ceci implique le résultat suivant.

Proposition 3.2.1 *Le problème* SAT FINI PO *est récursivement énumérable.*

Démonstration. Il suffit d'énumérer toutes les structures potentielles finies $\mathscr{S}$ et vérifier si $\mathscr{S} \models \varphi$. ■

Le résultat qui suit montre que le problème SAT FINI PO est indécidable. La preuve que l'on va en donner illustre un principe de codage du calcul d'une machine par une interprétation logique.

Théorème 3.2.2 (Trakhtenbrot) *Le problème* SAT PO FINI *est indécidable. En particulier,* SAT FINI PO *n'est pas co-récursivement énumérable.*

Démonstration. On va montrer que le problème est indécidable par réduction à partir du problème de l'arrêt d'une machine de Turing. Soit $M = (Q, \Gamma, \Sigma, \delta, q_0, q_a)$ une machine de Turing à un ruban. Sans perte de généralité, on supposera dans la suite que M s'arrête seulement si elle rencontre l'état acceptant q_a (i.e. qu'il n'y a pas d'état rejetant ou que la machineboucle indéfiniment si elle atteint celui-ci). On va construire une formule φ_M sur la signature $\sigma = \{0, <, Succ, Pred, (\text{Symb}_\gamma)_{\gamma \in \Gamma}, (\text{Etat}_q)_{q \in Q}, \text{Tete}\}$ où :

- $<$ sera interprété comme l'ordre naturel sur les entiers et $Succ$ et $Pred$ comme les fonctions successeur et prédécesseur compatible avec $<$.
- Chaque Symb_γ, $\gamma \in \Gamma$ ainsi que Tete est un symbole de prédicat binaire.
- Chaque Etat_q, $q \in Q$, est un prédicat unaire.

On veut établir l'équivalence suivante :

$$M \text{ s'arrête sur le mot vide en entrée} \iff \varphi_M \text{ a un modèle fini}$$

La formule $\varphi_{M,w}$ va simplement exprimer le comportement de la machine M sur w dans le temps. On a besoin de l'ordre (et des fonctions associées) pour parler des instants dans le temps : la formule devra aussi axiomatiser l'ordre pour s'assurer de son interprétation.

Dans la suite, on pose $Q = \{1, \ldots, |Q|\}$ (avec q_0 représenté par 1), $\Sigma = \{1, \ldots, |\Sigma|\}$, $\Gamma = \{1, \ldots, |\Gamma|\}$. De même, les mouvements de la machine de Turing sont pris dans l'ensemble $\{+1, -1, 0\}$. On suppose évidemment que la fonction de transition de la machine de Turing est totale : en dehors des état finaux, une transition est prévue quelque soit l'état et le symbole lus.

La formule $\Phi_{M,w}$ que l'on construit est la conjonction de quatre formules :

ORDRE, INIT, CONTRAINTE, TRANSITION$_M$, FINAL.

La formule ORDRE va axiomatiser l'ordre, le successeur et le prédécesseur. Les formules INIT et FINAL vont coder respectivement la configuration initiale de la machine et sa configuration finale ; CONTRAINTE est un ensemble de clauses qui doit être satisfaite par toute machine et TRANSITION$_M$ code les transitions de la machine M.

L'ordre $<$ La formule ORDRE sera la conjonction des formules suivantes.

- ⋆ $O_1 : \forall x \neg(x < x) \wedge \forall y \, (x = y \vee x < y \vee y < x) \wedge \forall z (x < y \wedge y < z \rightarrow x < z)$
  Le prédicat $<$ est un ordre total strict.
- ⋆ $O_2 : \forall x \, (x = 0 \vee x > 0)$
  La constante 0 est le plus petit élément pour $<$. On ne dit rien sur l'existence d'un plus grand élément éventuel.
- ⋆ $O_3 : \forall x \, ((\exists y \; x < y \rightarrow x < Succ(x)) \wedge \forall y \, \neg(x < y < Succ(x)))$
- ⋆ $O_4 : \forall x \; x \neq 0 \rightarrow Succ(Pred(x)) = x$
  Les deux formules O_3 et O_4 définissent les fonctions $Pred$ et $Succ$ comme attendues.

1. Régler les détails éventuels en exercice

Les conditions initiales La formule INIT va être la conjonction des formules suivantes.

- ⋆ $I_1 : \forall c\ \mathrm{Symb}_{\sqcup}(0,c)$
  Cette clause signifie que les cases du ruban sont toutes initialisées à $\sqcup$.
- ⋆ $I_2 : \mathrm{Etat}_{q_0}(0) \wedge \mathrm{Tete}(0, Succ(0))$
  A l'instant 0, on est dans l'état initial q_0 et la tête de lecture est sur la case 1.

Les contraintes de machines Dans cette partie, on exprime le fait que les prédicats $\mathrm{Symb}_\gamma, \mathrm{Etat}_q, \mathrm{Tete}$ codent bien le comportement d'une machine de Turing. La formule CONTRAINTE est la conjonction des formules suivantes.

- ⋆ $C_1 : \forall t \forall c\ \mathrm{Symb}_1(t,c) \vee \cdots \vee \mathrm{Symb}_{|\Gamma|}(t,c)$.
  A chaque instant t, chaque case c contient un symbole de Γ (possiblement $\sqcup$ ou dans $\Sigma \subseteq \Gamma$).
- ⋆ $C_2 : \forall t \forall c \bigwedge_{\gamma \neq \gamma'} \mathrm{Symb}_\gamma(t,c) \to \neg \mathrm{Symb}_{\gamma'}(t,c)$
  A chaque instant t, une case c ne peut contenir qu'un seul symbole γ.
- ⋆ $C_3 : \forall t (\mathrm{Etat}_1(t) \vee \cdots \vee \mathrm{Etat}_{|Q|}(t)) \wedge \bigwedge_{q \neq q'} \mathrm{Etat}_q(t) \to \neg \mathrm{Etat}_{q'}(t)$
  A chaque instant t, la machine est exactement dans un état q.
- ⋆ $C_5 : \forall t \exists c\ (\mathrm{Tete}(t,c) \wedge \forall c' \neq c\ \neg \mathrm{Tete}(t,c')$.
  A chaque instant t, la tête de lecture est dans exactement une seule case.
- ⋆ $C_7 : \forall t \forall c\ \mathrm{Tete}(t,c) \wedge (\exists t'\ t' > t) \to \forall c' \neq c\ (\bigwedge_{\gamma \in \Gamma} \mathrm{Symb}_\gamma(t,c') \to \mathrm{Symb}_\gamma(Succ(t),c'))$.
  Si la tête de lecture est sur la case c à l'instant t (et que t n'est pas le dernier instant du temps), le contenu des autres cases $c' \neq c$ ne sera pas modifié à l'instant d'après.

Les transitions de la machine Cette partie est dépendante de la machine M de départ. On va coder par des formules les transitions de la machine. On rappelle que pour une machine la fonction de transition δ est de la forme :

$$\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{+1, -1, 0\}.$$

Une formule par transition sera nécessaire. On construit $\mathrm{TRANSITION}_M$ la conjonction de formules du même type suivant pour toute transition $\delta(q,\gamma) = (q', \gamma', d)$ de la machine telle que $q \neq q_a$.

$$T_1 : \quad \forall t \forall c \mathrm{Tete}(t,c) \wedge \quad \mathrm{Etat}_q(t) \wedge \mathrm{Symb}_\gamma(t,c) \longrightarrow$$

$$\mathrm{Tete}(Succ(t), c+d) \wedge \mathrm{Etat}_{q'}(Succ(t)) \wedge \mathrm{Symb}_{\gamma'}(Succ(t), c).$$

où $c+d$ est $Succ(c)$ si $d = +1$, $Pred(c)$ si $d = -1$, et c si $d = 0$.

La signification en est : si à un instant t, la tête de lecture est sur la case i que l'on lit le symbole γ sur cette case et que l'on est dans l'état q (sauf si q est l'état final) alors, le calcul se poursuit et à l'instant suivant $Succ(t)$, on a pour conséquence sur l'état, le symbole écrit et la position de la tête le résultat de la transition.

La configuration finale Il nous reste a décrire le fait que si, à un certain instant du temps t, la machine est dans l'état acceptant, alors elle va s'arrêter. On le dit indirectement en indiquant que t est le dernier élément.

$$\mathrm{FINAL} \equiv \forall t\ \mathrm{Etat}_{q_a}(t) \longrightarrow \forall t'\ (t' < t \vee t = t')$$

.

Si M s'arrête sur le mot vide alors il existe un instant t où M atteint l'état acceptant. Un modèle fini φ_M (de fait, à isomorphisme près, il n'y a qu'un seul modèle fini pour φ_M) est facilement constructible à partir du diagramme espace/temps de la machine de Turing [2]. Réciproquement, si φ_M a un modèle

2. Formellement, on peut le montrer par induction en supposant que si $C_1, \ldots, C_t$ sont t configurations consécutives de calcul de la machine et que toutes les variables $\mathrm{Etat}(t,-)$, $\mathrm{Tete}(t,-)$ et $\mathrm{Case}(t,-,-)$ ont des valeurs de vérité compatibles avec ces descriptions alors pour toute configuration possible C_{t+1}, il existe une extension possible (unique qui plus est si on enrichit un peu la formule) de l'assignation de valeur de vérité aux variables indicées en "t+1".

fini, celui-ci vérifie les formules constituants CONTRAINTE donc il correspond a un calcul de Machine. Comme il vérifie INIT et FINAL, il s'agit donc d'un calcul de machine de Turing partant sur l'entrée vide et finissant dans un état acceptant. Enfin, TRANSITION_M spécifie que la machine n'est pas quelconque mais qu'il s'agit de M. Enfin, pour coclure, la construction de la formule φ_M à partir de M est clairement récursive. ■

Remarque. Le résultat d'indécidabilité ci-dessus reste vrai pour toute signature fixée σ contenant au moins un symbole de relation binaire.

Exercice 17 Redémontrer le théorème 3.2.2 en prenant comme modèle de calcul de départ pour M une machine à registre ou machine RAM) à programme goto.

Considérons maintenant le problème suivant.

VALID PO
Entrée : Une formule close du premier-ordre φ sur une signature σ
Question : φ est-elle valide ?

Corollaire 3.2.3 (Church) *Le problème* VALID PO *est indécidable.*

Démonstration. Dans la preuve du Théorème 3.2.2, on voit facilement que la formule φ_M a un modèle infini si et seulement si M ne s'arrête pas sur le mot vide en entrée. Soit Ψ la formule suivante :

$$\Psi \equiv \forall t \exists t' \; t < t'.$$

La formule $\varphi_M \wedge \Psi$ a alors un modèle si et seulement si M ne s'arrête pas. Autrement dit, $\neg(\varphi_M \wedge \Psi)$ est valide si et seulement si M s'arrête. ■

Noter que le résultat précédent nous permet aussi de montrer (par réduction) que le problème VALID PO n'est pas co-récursivement énumérable. Par contre, par utilisation du théorème de complétude et énumération des preuves, on peut voir que VALID PO est récursivement énumérable. De façon surprenante, la restriction aux structures finies amène à une situation duale du cas général :
- Le problème SAT FINI PO est récursivement énumérable mais n'est pas co-récursivement énumérable
- Le problème SAT PO est co-r.e. mais pas r.e.

3.3 Méthodes de décision : les jeux d'Ehrenfeucht-Fraïssé

Dans cette section, on va s'intéresser à un outil classique permettant entre autres de montrer la décidabilité de certaines théories logiques.

3.3.1 Jeux et va-et-vient

Dans la suite $\mathscr{A}$ et $\mathscr{B}$ désignent deux structures sur un certain vocabulaire σ. On appelera $c_1^{\mathscr{A}},\dots,c_k^{\mathscr{A}}$ les constantes de $\mathscr{A}$ et $c_1^{\mathscr{B}},\dots,c_k^{\mathscr{B}}$ les constantes de $\mathscr{B}$. Soient $a_1,\dots,a_n$ des éléments de (du domaine de) $\mathscr{A}$ et, de même, $b_1,\dots,b_n$ des éléments de $\mathscr{B}$. On dit que l'application f qui à chaque a_i associe b_i (pour $i \leq n$) est un *isomorphisme partiel* entre $\mathscr{A}$ et $\mathscr{B}$ si :

- $\forall i,j \leq n,\ a_i = a_j$ ssi $b_i = b_j$,
- $\forall i \leq n,\ \forall c \in \sigma,\ a_i = c^{\mathscr{A}}$ ssi $b_i = c^{\mathscr{B}}$,
- pour toute relation $R \in \sigma$ d'arité k :

$$\forall i_1,\dots,i_k \leq n,\ R^{\mathscr{A}}(a_{i_1},\dots,a_{i_k}) \text{ ssi } R^{\mathscr{B}}(b_{i_1},\dots,b_{i_k}).$$

Définition 3.3.1 *Le jeu d'Ehrenfeucht-Fraïssé à n coups sur deux structures $\mathscr{A}$ et $\mathscr{B}$ est un jeu à deux joueurs* I *et* II *dont les règles sont les suivantes.*

- *Au coup i (pour $i \in \{1,\dots,n\}$), le joueur* I *choisit l'une des structures $\mathscr{A}$ ou $\mathscr{B}$ et un élément, dans cette structure. Le joueur* II *répond en choisissant un élément dans l'autre structure. On note a_i l'élément choisi dans $\mathscr{A}$ et b_i l'élément choisi dans $\mathscr{B}$.*
- II *gagne si l'application $a_i \mapsto b_i$, pour $i = 1,\dots,n$ et $c_j^{\mathscr{A}} \mapsto c_j^{\mathscr{B}}$, pour $j = 1,\dots,k$, est un isomorphisme partiel. Dans tous les autres cas, c'est* I *qui gagne.*

En d'autres termes, II cherche à montrer que les deux structures sont similaires et I qu'elles sont différentes.

Notation : on note $\mathscr{A} \equiv_n \mathscr{B}$ le fait que II a une stratégie gagnante dans le jeu à n coups sur $\mathscr{A}$ et $\mathscr{B}$.

On peut proposer une formulation équivalente (facile à démontrer) en termes dit de *va-et-vient*.
- $\mathscr{A} \equiv_0 \mathscr{B}$ si les restrictions de $\mathscr{A}$ à $\mathscr{B}$ à leurs ensembles de constantes respectifs sont partiellement isomorphes [3].
- $\mathscr{A} \equiv_{n+1} \mathscr{B}$ si :
  - pour tout a dans $\mathscr{A}$, il existe b dans $\mathscr{B}$ t.q. $(\mathscr{A}, a) \equiv_n (\mathscr{B}, b)$ [4],
  - pour tout b dans $\mathscr{B}$, il existe a dans $\mathscr{A}$ t.q. $(\mathscr{A}, a) \equiv_n (\mathscr{B}, b)$ [5].

3.3.2 Formules et types

Définition 3.3.2 *La profondeur (ou le rang) de quantification $qr(\varphi)$ d'une formule du premier-ordre φ se définit de la façon suivante :*
- *si φ est atomique, $qr(\varphi) = 0$,*
- *$qr(\varphi_1 \wedge \varphi_2) = qr(\varphi_1 \vee \varphi_2) = max(qr(\varphi_1), qr(\varphi_2))$,*
- *$qr(\varphi) = qr(\neg\varphi)$,*
- *$qr(\exists x\varphi) = qr(\forall x\varphi) = qr(\varphi) + 1$.*

Exemple. Pour la formule $\varphi : \exists x \forall y R(x, y) \vee \forall x \forall z \exists t S(x, x, y, t)$, on a $\mathrm{qr}(\varphi) = 3$

Dans la suite, on dira aussi que deux formules $\varphi(\overline{x})$ et $\psi(\overline{x})$ avec $\overline{x} = (x_1,\dots,x_n)$ sont *logiquement équivalentes* (on devrait rajouter *finiement*)si pour toute structure finie $\mathscr{A}$ et tout uplet $\overline{a} = (a_1,\dots,a_n)$ d'éléments de $\mathscr{A}$, on a :

$$\mathscr{A} \models \varphi(\overline{a}) \text{ ssi } \mathscr{A} \models \psi(\overline{a}).$$

Définition 3.3.3 *Soit $\mathscr{A}$ une σ-structure et $\overline{a} = (a_1,\dots,a_n)$ un uplet d'éléments de $\mathscr{A}$. Le n-type de rang k sur $(\mathscr{A}, \overline{a})$ est constitué des formules à n variables libres de profondeur k vraies dans $(\mathscr{A}, \overline{a})$, plus précisément :*

$$type_k(\mathscr{A}, \overline{a}) = \{\varphi(\overline{x}) : qr(\varphi) \text{ et } \mathscr{A} \models \varphi(\overline{a})\}.$$

Par extension un n-type de rang k est un ensemble de formules de la forme $type_k(\mathscr{A}, \overline{a})$ pour un certain $\mathscr{A}$ et un certain $\overline{a}$.

Théorème 3.3.4 *On suppose que σ est une signature finie, k et n entiers fixés.*

1. *À équivalence logique près, il n'y a qu'un nombre fini de formules du premier-ordre φ t.q. $qr(\varphi) = k$ et φ a n variables libres.*
2. *Le nombre de n-types de rang k est fini.*

Démonstration. **1)** Par récurrence sur k pour tout n. On commence par le montrer pour $k = 0$. Toute formule sans quantificateur $\varphi(\overline{x})$ est combinaison booléenne de formules atomiques portant :
- sur les variables libres (au plus n),
- sur les constantes de σ (qui est finie).

---
3. si l'application qui associe pour chaque constante ses interprétations respectives dans $\mathscr{A}$ et $\mathscr{B}$ est un isomorphisme partiel
4. Le *va* (*forth* en anglais)
5. Le *vient* (*back* en anglais)

On peut normaliser de telles formules et les considérer sous la forme :

$$\bigvee_{i\in I} C_i$$

où chaque C_i est une conjonction de formules atomiques. Il n'y a qu'un nombre fini de telles clauses possibles dû au caractère fini du langage et donc un nombre fini de façon de les combiner[6]. On suppose la propriété vraie au rang k et on la montre pour $k+1$. Toute formule $\varphi(\overline{x})$ de profondeur $k+1$ à n variables libres est combinaison booléenne de formules de la forme $\exists y\psi(\overline{x}, y)$ où ψ est de profondeur k et a $n+1$ variables libres. Par hypothèse, le nombre de formules de profondeur k à $n+1$ variables libres est fini, à équivalence logique près. Soient $\varphi_1(\overline{x}, y), \ldots, \varphi_K(\overline{x}, y)$ ces formules. Alors, pour tout $\psi(\overline{x}, y)$, il existe $i \leq K$ tel que pour toute structure $\mathscr{A}$ et tout $\overline{a}$, a' dans $\mathscr{A}$:

$$\mathscr{A} \models \psi(\overline{a}, a') \text{ ssi } \mathscr{A} \models \phi_i(\overline{a}, a').$$

On a alors, immédiatement :

$$\mathscr{A} \models \exists y\psi(\overline{a}, y) \text{ ssi } \mathscr{A} \models \exists\phi_i(\overline{a}, y).$$

Le nombre de telles formules $\exists y\psi(\overline{x}, y)$ est bien fini et il en est de même de leur combinaison booléenne à équivalence logique près.

2) D'après la première partie, on sait que le nombre de formules de profondeur k et à n variables libres est fini à équivalence logique près. Soit $\varphi_1(\overline{x}), \ldots, \varphi_N(\overline{x})$ une énumération de ces formules.

Si un ensemble de formules T est un n-type de rang k alors il existe $\mathscr{A}$ et $\overline{a}$ uplet d'éléments de $\mathscr{A}$ tels que $T = \text{type}_k(\mathscr{A}, \overline{a})$. Examinons l'appartenance de chaque $\varphi_i(\overline{x})$ à T. Soit $I \subseteq N = \{1, \ldots, N\}$ l'ensemble d'indices tels que : $\forall i \in I, \varphi_i(\overline{x}) \in T$ et $\forall i \in N\backslash I, \varphi_i(\overline{x}) \notin T$. Pour toute formule $\varphi(\overline{x})$ de rang k, soit $\varphi_i(\overline{x})$ son équivalent logique dans la liste établie, on a :

$$\varphi(\overline{x}) \in T \iff \mathscr{A} \models \varphi(\overline{a}) \iff \mathscr{A} \models \varphi_i(\overline{a}) \iff \varphi_i(\overline{x}) \in T$$

L'ensemble T contient donc toutes les formules logiquement équivalentes à une des formules $\varphi_i(\overline{x})$ pour $i \in I$ et aucune des formules logiquement équivalentes à une des formules $\varphi_i(\overline{x})$ pour $i \notin I$. Chaque k-type de rang n est donc complétement décrit par la liste des formules $\varphi_1(\overline{x}), \ldots, \varphi_N(\overline{x})$ qui lui appartiennent et ne lui appartiennent pas. En d'autres termes, les 2^N formules,

$$\text{DESC}_I(\overline{x}) = \bigwedge_{i\in I} \varphi_i(\overline{x}) \wedge \bigwedge_{i\notin I} \neg\varphi_i(\overline{x}),$$

décrivent tous les n-types de rang k possibles.

On reprend le dernier argument de la preuve précédente dans l'énoncé du corollaire suivant.

Corollaire 3.3.5 *Soit $T_1, \ldots, T_N$ une énumération des n-types de rang k et $\text{DESC}_1(\overline{x}), \ldots, \text{DESC}_K(\overline{x})$ les formules décrivant ces n-types, alors, pour toute structure $\mathscr{A}$, tout $\overline{a}$ dans $\mathscr{A}$ et tout $i \leq N$:*

$$\mathscr{A} \models \text{DESC}_I(\overline{a}) \iff T_i = \textit{type}_k(\mathscr{A}, \overline{a}),$$

où i et l'indice de la partition I dans l'énumération.

Remarquer que chaque formule $\text{DESC}_K(\overline{x})$ est de profondeur k. On peut maintenant énoncer et prouver le résultat principal de cette partie.

Théorème 3.3.6 (Fraïssé, Ehrenfeucht) *Soit σ une signature relationnelle. Soient $\mathscr{A}$ et $\mathscr{B}$ deux σ-structures alors, pour tout k fixé :*

$$\mathscr{A} \equiv_k \mathscr{B} \textit{ ssi } \mathscr{A} \textit{ et } \mathscr{B} \textit{ vérifient les mêmes formules de profondeur } k.$$

6. On peut aussi remarquer que considérer les formules à équivalence logique près, permet d'éliminer toute clause impliquée par d'autres. Chaque C_i peut donc être vue comme une description exhaustive en fonction de σ, $\overline{x}$ et des constantes. Essayez de donner toutes ces descriptions pour $\sigma = \{R, c\}$, où R est une relation binaire, c une constante et $\overline{x} = (x_1, x_2)$.

Démonstration. Par induction sur k. On remarque au préalable, pour $k = 0$, que $\mathscr{A} \equiv_0 \mathscr{B}$ ssi les restrictions de $\mathscr{A}$ et $\mathscr{B}$ à leur ensemble de constantes respectifs sont isomorphes et donc ssi les deux structures satisfont les mêmes formules atomiques donc les mêmes formules de profondeur 0 (i.e. sans quantificateurs).

On prouve la première implication (gauche-droite) en supposant qu'elle est vraie au rang k et que l'on a $\mathscr{A} \equiv_{k+1} \mathscr{B}$. Toute formule de rang $k+1$ est combinaison booléenne de formules de la forme $\exists x\varphi(x)$ avec $\text{qr}(\varphi) = k$. Pour chaque formule de ce type, si $\mathscr{A} \models \exists x\varphi(x)$ alors, il existe a dans $\mathscr{A}$ tel que $\mathscr{A} \models \varphi(a)$. Comme $\mathscr{A} \equiv_{k+1} \mathscr{B}$, il existe b t.q. $(\mathscr{A}, a) \equiv_k (\mathscr{B}, b)$. Par hypothèse d'induction, $(\mathscr{A}, a)$ et $(\mathscr{B}, b)$ vérifient les mêmes formules de profondeur k donc $\mathscr{B} \models \varphi(b)$ d'où $\mathscr{B} \models \exists x\varphi(x)$. On montre le passage inverse ($\mathscr{B} \models \exists x\varphi(x)$ implique $\mathscr{A} \models \exists x\varphi(x)$) de façon similaire. On conclut en remarquant que puisque $\mathscr{A}$ et $\mathscr{B}$ vérifient les mêmes formules de la forme $\exists x\varphi(x)$ avec $\text{qr}(\varphi) = k$, il en est de même pour toute combinaison booléenne de ces formules.

Pour la seconde implication, on suppose qu'elle est vraie au rang k et que $\mathscr{A}$ et $\mathscr{B}$ vérifient les mêmes formules de profondeur $k+1$. Soit a quelconque dans $\mathscr{A}$ et $\text{type}_k(\mathscr{A}, a)$ le 1-type de rang k pour a et $\mathscr{A}$. Soit $\text{DESC}_i(x)$ la description de $\text{type}_k(\mathscr{A}, a)$. On sait que $\text{qr}(\text{DESC}_i(x)) = k$. On a $\mathscr{A} \models \text{DESC}_i(a)$ donc $\mathscr{A} \models \exists x\text{DESC}_i(x)$. On a maintenant $\text{qr}(\exists x\text{DESC}_i(x)) = k+1$ et on sait, par hypothèse, que $\mathscr{A}$ et $\mathscr{B}$ vérifient les mêmes formules de profondeur $k+1$. Donc, $\mathscr{B} \models \exists x\text{DESC}_i(x)$. Soit b un témoin t.q. $\mathscr{B} \models \text{DESC}_i(b)$. Ceci implique que $\text{type}_k(\mathscr{A}, a) = \text{type}_k(\mathscr{B}, b)$ donc, pour toute formule de rang k, φ, on a :

$$(\mathscr{A}, a) \models \varphi \iff (\mathscr{B}, b) \models \varphi.$$

Donc $(\mathscr{A}, a)$ et $(\mathscr{B}, b)$ satisfont les mêmes formules de profondeur k donc, par hypothèse de récurrence, $(\mathscr{A}, a) \equiv_k (\mathscr{B}, b)$. Ceci conclut la preuve pour le *va* (i.e. partant de $\mathscr{A}$ et a quelconque). Le *vient* se prouve de façon similaire. On obtient $\mathscr{A} \equiv_{k+1} \mathscr{B}$

Le théorème de Fraïssé-Ehrenfeucht établit une correspondance entre définissabilité au premier-ordre (du moins, il semble) et k-équivalence ($\equiv_k$) de structures. Paradoxalement, c'est l'implication la plus facile à prouver du théorème de Fraïssé-Ehrenfeucht qui est la plus souvent utilisée. Les jeux vont nous servir à montrer que certaines propriétés ne sont pas définissables au premier-ordre (alors que pour montrer la définissabilité, il est souvent plus facile d'exhiber directement une formule) mais aussi que certains problèmes de décision logique sont décidables (en montrant que l'existence d'un modèle est subordonnée à l'exsitence d'un petit modèle).

3.3.3 Théorie du premier ordre monadique

Soit $\mathscr{R}el_1^\omega$ la classe de toutes les structures monadiques i.e. ne contenant que des prédicats unaires.

SAT FINI($\mathscr{R}el_1^\omega$)
Entrée : Une formule du premier-ordre φ sur une signature $\sigma = \{U_1, ..., U_k\}$ où chaque U_i est un symbole de relation unaire.
Question : φ a-t-elle un modèle (fini ou non) ?

Soient $\mathscr{A} = \langle D, U_1^{\mathscr{A}}, ..., U_k^{\mathscr{A}}\rangle$ notée $\mathscr{A} = \langle D, U_1, ..., U_k\rangle$ et $\mathscr{B} = \langle D', U_1^{\mathscr{B}}, ..., U_k^{\mathscr{B}}\rangle$ notée $\mathscr{B} = \langle D', V_1, ..., V_k\rangle$ deux structures sur une signature σ ne contenant que des symboles de relations unaires. Soit $\epsilon = (\epsilon_1, \ldots, \epsilon_k) \in \{-1, +1\}^k$. On note $\epsilon(\overline{U})$ l'ensemble $\bigcap_{i \le k} U_i^{\epsilon_i}$ avec la convention :

$$U_i^{+1} = U_i \text{ et } U_i^{-1} = D \backslash U_i.$$

Soit $q \in \mathbb{N}^*$. Soient m, n deux entiers naturels, on notera $m =_q n$ si :
– Soit $m < q$, $n < q$ et $m = n$ ou,
– $m \ge q$ et $n \ge q$.
On note (H_q) la propriété suivante :

$$(H_q) : \text{ Pour tout } \epsilon \in \{-1, +1\}^k, card(\epsilon(\overline{U})) =_q card(\epsilon(\overline{V})).$$

Cette propriété impose que chaque sous-ensemble de la partition induite par $\overline{U}$ et le sous-ensemble correspondant de la partition induite par $\overline{V}$ sont de même taille si celle-ci est plus petite que q.

Proposition 3.3.1 *Pour tout $q \in \mathbb{N}$, si deux structures $\mathscr{A} = \langle D, U_1^{\mathscr{A}}, ..., U_k^{\mathscr{A}} \rangle$ et $\mathscr{B} = \langle D', U_1^{\mathscr{B}}, ..., U_k^{\mathscr{B}} \rangle$ vérifient la propriété (H_q) alors :*

$$\mathscr{A} \equiv_q \mathscr{B}$$

i.e. $\mathscr{A}$ et $\mathscr{B}$ vérifient les même formules du premier-ordre de profondeur q.

Démonstration. La preuve est élémentaire. Supposons que le joueur I joue un élément $a_i \in \mathscr{A}$ à l'étape $i \leq q$. Supposons que a_i soit différent de $a_1, ..., a_{i-1}$. Si ce n'est pas le cas, i.e. si $a_i = a_j$ pour $j < i$, II peut rejouer l'élément b_j correspondant. Soit ϵ tel que $a_i \in \epsilon(\overline{U})$ et soit t le nombre d'éléments dans $\epsilon(\overline{U})$ parmi $a_1, ..., a_{i-1}$ (bien sûr, les t éléments correspondants de $\mathscr{B}$ sont dans $\epsilon(\overline{V})$). Comme $t \leq i-1 \leq q$, comme $\mathscr{A}$ et $\mathscr{B}$ vérifient (H_q), on a $\mathsf{card}\,\epsilon(\overline{V}) \geq t+1$. Le joueur II répond en jouant un des éléments de $\epsilon(\overline{U})$ non joués jusque là. L'argument est similaire si I joue dans $\mathscr{B}$ à l'étape i. ■

Le résultat précédent a pour corollaire que toute formule monadique a un modèle (fini ou infini) si elle a un modèle de cardinalité inférieure à $q2^k$ où q est la profondeur en quantificateur et k le nombre de prédicats monadiques. On en déduit le résultat suivant.

Proposition 3.3.2 *Le problème* SAT FINI($\mathscr{R}el_1^{\omega}$) *est décidable*

Démonstration. En vertu de la remarque précédente, pour déterminer si une formule φ (à q variables et k prédicats) a un modèle, il suffit d'examiner si elle a un modèle de cardinalité bornée par $q2^k$. On peut le faire par recherche exhaustive. ■

3.3.4 Le cas d'une seule fonction unaire

On peut montrer que le problème de décider si une formule du premier-ordre sur une signature contenant un seul symbole de fonction unaire est décidable. La preuve en est un peu longue. On va s'intéresser au problème plus simple suivant.

SAT FINI($\mathscr{B}ij$)
Entrée : une formule φ close du premier ordre sur une signature $\sigma = \{f\}$ où f est un symbole de fonction unaire
Question : φ a-t-elle un modèle tel que f est une fonction bijective

Soit $q \in \mathbb{N}$. Soit $\mathscr{A} = \langle D, f \rangle$ une structure finie où f est bijective (on parle de structure bijective). On appelle cycle de longueur n de $\mathscr{A}$, un ensemble d'éléments $\{x_1, ..., x_n\}$ de D tel que $f(x_1) = x_2$, $f(x_2) = x_3$, ..., $f(x_n) = x_1$.

Soient $\mathscr{A}$ et $\mathscr{B}$ deux σ-structures et soit C_q la propriété suivante :

- pour tout $k \leq 2^{q+1}$, $\mathscr{A}$ et $\mathscr{B}$ ont le même nombre n de cycles de tailles k si $n \leq q$ ou tous les deux plus de q cycles.
- $\mathscr{A}$ et $\mathscr{B}$ ont le même nombre n de cycles de tailles supérieures à 2^{q+1} si $n \leq q$ ou tous les deux plus de q cycles.

Proposition 3.3.3 *Soint $\mathscr{A}$ et $\mathscr{B}$ deux $\{f\}$-structures bijectives vérifiant la propriété (C_q). Alors :*

$$\mathscr{A} \equiv_q \mathscr{B}$$

Démonstration. La preuve s'inspire de celle de la proposition 3.3.1 sur certains points. Le nombre de"petit" cycles d'une taille donnée dans chaque structure est identique jusqu'à q ou différents mais plus grand que q dans les deux structures. Comme $q \geq i$, pour tout coup joué par I dans un de ces petits cycles, II peut trouver une réponse adéquate (i.e. à chaque coup de I portant dans un nouveau petit cycle, II peut choisir un petit cycle non joué dans l'autre structure).

Le seul problème à résoudre est si $\mathscr{A}$ a un cycle de grande taille $m \geq 2^{q+1}$ mais que $\mathscr{B}$ n'a pas de cycle de cette taille. Comme la condition (C_q) est satisfaite, $\mathscr{B}$ a au moins un cycle de taille $m' \geq 2^{q+1}$ (avec $m \neq m'$). Examinons ce que donne le jeu sur ces deux cycles respectifs (on considère dans la suite que $\mathscr{A}$ et $\mathscr{B}$ se réduisent au graphe *orienté* induit par ces grands cycles).

Soit d la distance induite par la relation de graphe i.e. pour x et y dans $\mathscr{A}$ ou $\mathscr{B}$, $d(x,y)=n$ si le plus petit chemin entre x et y ou entre y et x est de longueur n. On va montrer, par induction, que le joueur II peut jouer de telle manière qu'à chaque tour $i \leq q$, il maintient vrai la propriété suivante. Pour tout $j, j' \leq i, j \neq j'$:

$$(*)_i \left\{ \begin{array}{l} \text{si } d(a_j, a_{j'}) \leq 2^{q-i} \text{ alors } d(a_j, a_{j'}) = d(b_j, b_{j'}), \\ d(a_j, a_{j'}) > 2^{q-i} \iff d(b_j, b_{j'}) > 2^{q-i}. \end{array} \right.$$

Pour $i=2$, il est facile de voir que pour m assez grand, la propriété $(*)_2$ est satisfaite. Supposons que la propriété est vraie au rang $i<q$. Supposons que I joue un élément a_{i+1} dans $\mathscr{A}$ (l'autre cas est similaire).

Soient $A_i = \{a_1, \dots, a_i\}$ et $B_i = \{b_1, \dots, b_i\}$ les ensembles d'éléments déjà joués jusqu'au coup i. Si a_{i+1} a déjà été joué, II répond par l'élémént correspondant de $\mathscr{B}$. On peut donc supposer que $a_{i+1} \neq a_l$, $\forall l \leq i$. On appelle $N^{i+1}(a_{i+1})$ l'ensemble des points à distance $2^{k-(i+1)}$ de a_{i+1}, appelés le voisinage de a_{i+1} à distance $2^{k-(i+1)}$. Soit $S_i = A_i \cap N^{i+1}(a_{i+1})$ les points de A_i qui sont dans ce voisinage.

Si $S_i \neq \emptyset$, soit $S'_i \subseteq B_i$ le sous-ensemble des points de B_i correspondants. Soient $a_j \in S_i$, $b_j \in S'_i$ et $b \in \mathscr{B}$, l'unique point de $\mathscr{B}$ tel qu'il y a un chemin de a_{i+1} à a_j et de b à b_j (ou de a_j à a_{i+1} et de b_j à b) et $d(a_{i+1}, a_j) = d(b, b_j)$. On pose $b = b_{j+1}$. Par définition du voisinage, pour tout $a_j, a_l \in S_i$ et b_j, b_l dans S'_i, on a $d(a_j, a_l) \leq 2^{q-i}$ et $d(b_j, b_l) \leq 2^{q-i}$. Donc, par hypothèse de récurrence $d(a_j, a_l) = d(b_j, b_l)$. Ceci implique que pour tout $a_j \in S_i$ et b_j dans S'_i, $d(a_j, a_{i+1}) = d(b_j, b_{i+1}) < 2^{q-(i+1)}$ De même, pour tout $a_j \in A_i \backslash S_i$, $d(a_j, a_{i+1}) > 2^{q-(i+1)}$ et $d(b_j, b_{i+1}) > 2^{q-(i+1)}$.

Si $S_i = \emptyset$, a_{i+1} est à distance au moins $2^{q-(i+1)}$ de tous les points de A_i. Il y a forcément un point b de B à distance au moins $2^{q-(i+1)}$ de tous les points de B_i : le nombre de points maximal dans un voisinage de rayon $2^{q-(i+1)}$ est $2^{q-i}+1$; il y a i tels voisinages à considérer du fait des coups précédents ; donc au plus $i2^{k-i}+1$ sont inutilisables. Mais comme la taille du cycle est $2^{q+1} \geq (i2^{q-i}+1)$, quelque soit $i = 0, \dots, q$, on peut trouver un tel b. On prend alors $b_{i+1} = b$ et la propriété $(*)_{i+1}$ est bien vérifiée.■

Proposition 3.3.7 *Le problème $\mathscr{B}ij$ est décidable.*

Démonstration. Soit φ une formule du premier ordre sur un langage unaire de profondeur q. On sait que si deux structures vérifient la propriété C_q alors elles sont indiscernables. On voit facilement que si $\mathscr{A}$ est une structure de taille supérieure à $(2^{q+2}+1)(q+1)$ alors il existe une structure $\mathscr{B}$ de taille inférieure ou égale à $(2^{q+2}+1)(q+1)$ telle que :

$$\mathscr{A} \equiv_q \mathscr{B}.$$

En effet, $(2^{q+2}+1)(q+1)$ est la taille maximale d'une structure ayant $q+1$ cycles de tailles bornées par 2^{q+1} et $q+1$ cycle de tailles exactement $2^{q+1}+1$ (taille à laquelle on peut se restreindre pour les grand cycle en vertu du résultat précédent). Si φ a un modèle fini, elle en a donc un de taille inférieure à $(2^{q+2}+1)(q+1)$. On peut alors, par recherche exhaustive jusqu'à cette borne, déterminer si φ a un modèle. ■

3.3.5 Théorie de l'ordre total discret

On s'interesse à la théorie du premier-ordre de signature $\sigma = \{<, \min, \max\}$ d'une relation d'ordre total $<$ avec plus petit et plus grand éléments respectivement min et max. On peut montrer par des arguments de jeux la proposition suivante.

Proposition 3.3.4 *Supposons* $(E, <, \min, \max)$ *et* $(E', <', \min', \max')$ *deux ensembles totalement ordonnés. On a, pour tout entier k fixé :*

$$(E, <, \min, \max) \equiv_k (E', <', \min', \max') \textit{ ssi } |E| = |E'| \textit{ à } 2^k+1 \textit{ éléments près.}$$

Exercice 18 Déduire de la proposition précédente que la théorie du premier ordre d'une relation d'ordre total $<$ avec plus petit et plus grand éléments est décidable i.e. qu'étant donné une formule du p.o. φ sur $\sigma = \{<, \min, \max\}$ on peut décider si φ est vraie dans une structure ordonnée.

Montrer (lorsque vous aurez vu le chapitre sur la complexité), que ce problème de décision est dans PSPACE.

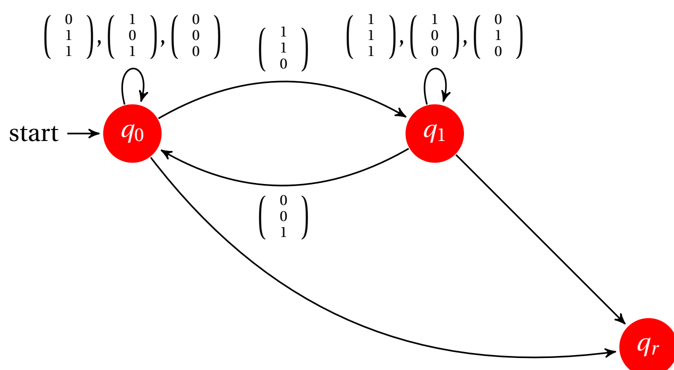

FIGURE 3.1 – L'automate d'addition

3.4 L'arithmétique de Presburger

On considère ici des formules du premier ordre sur la signature $\sigma = \{0, <, +\}$. Etant donné une σ-formule φ on cherche à déterminer si φ est vrai dans $\mathbb{N}$. On parle d'arithmétique de Presburger pour ce formalisme. La multiplication étant absente, on va voir que ce problème devient décidable.

Théorème 3.4.1 *Etant donné φ une formule du premier ordre sur $\sigma = \{0, <, +\}$, on peut décider si $\mathbb{N} \models \varphi$.*

Remarque. On peut prouver ce résultat de diverses manières. Une approche classique consiste à étendre le langage par les prédicats $n|$ pour tout n, interprété par $n|x$ ssi n divise x et à montrer que le langage obtenu admet l'élimination des quantificateurs au premier ordre. On conclut après en traitant directement les formules sans quantificateur. Une deuxième approche, plus récente, utilise la théorie des automates. C'est de celle-ci dont on s'inspirera (sans toutefois supposer de prérequis sur les automates)

Démonstration. Dans la suite, on considèrera des automates comme modèles de calcul, c'est à dire des machines de Turing à une seule bande dont la tête de lecture est unidirectionnelle (i.e. gauche droite) et en lecture seule et qui s'arrête dans un état soit acceptant, soit refusant après avoir lu la dernière lettre du mot. La table de transition d'une telle machine est donc de la forme : $\delta : Q \times \Sigma \to Q$.

Soit $\varphi = Q_1 x_1 Q_2 x_2 ..., Q_n x_n \Phi(x_1, ..., x_n)$, avec $Q_i \in \{\forall, \exists\}$ pour $i = 1, ..., n$. On pose :

$$\varphi_i(x_1, ..., x_i) = \exists x_{i+1} ... \exists x_n \Phi.$$

On commence par montrer le résultat pour φ_n qui est sans quantificateurs.
Reconnaitre une combinaison Booléenne de "contraintes" additives. On montre d'abord qu'étant donnés a_1, a_2, a_3, on peut contruire un automate tel que $a_1 + a_2 = a_3$. Pour se faire, il faut que l'entrée soit présentée dans $\Sigma^3 = \{0, 1\}^3$, bit par bit. Chaque symbole sera de fait un triplet (i_1, i_2, i_3) où i_1, i_2, i_3 sont, pour un i donné, les $i$èmes bits de a_1, a_2, a_3. L'algorithme vérifie simplement que l'addition s'effectue correctement bit par bit en tenant compte de la propagation de retenue éventuelle (voir la figure 3.1)..

Plus généralement, Soit $a_1, ..., a_n$ entiers et $\varphi_n(a_1, ..., a_n)$ une combinaison Booléenne de contraintes additives à vérifier. Comme-ci dessus, $a_1, ..., a_n$ vont être codés par une suite de symboles de Σ^n, chaque symbole représentant la suite des n bits d'indices i, pour $i \leq m = \max(|a_1|, \ldots, |a_n|)$. Chaque formule atomique A_j est de la forme $a_{j_1} + a_{j_2} = a_{j_3}$. Soit k le nombre de formules atomiques. Pour chaque formule atomique A_j à vérifier, on note $\delta_j : Q_j \times \Sigma^n \mapsto Q_j$ la fonction de transition obtenue par l'algorithme précédent. Pour connaitre, l'ensemble des valeurs de vérités pour les formules A_j il suffit de prendre le produit des fonctions de transitions :

$$\delta : Q_1 \times \cdots \times Q_k \times \Sigma^n \mapsto Q_1 \times \cdots \times Q_k$$

défini par, pour $q_i \in Q_i$, $s \in \Sigma^n$:

$$\delta(q_1,...,q_k,s) = (\delta_1(q_1,s),\ldots,\delta_k(q_k,s)).$$

Soit q_F l'état final du calcul. On pose $v_j = 1$ si $\pi_j^k(q_F)$ est acceptant et 0 sinon. Dans ce cas, la formule $\varphi_n(a_1,...,a_n)$ est satisfaite si la formule propositionnelle $\varphi(A_j/v_j)$ est vraie. Plus généralement, l'ensemble des assignations satisfaisant $\varphi(A_j)$ permet de déterminer l'ensemble des états acceptants.
Prise en compte de la quantification. On cherche maintenant à donner un algorithme $M' = (Q',\Sigma^i,\delta')$ pour la formule $\exists x_{i+1}\varphi_{i+1}(a_1,...,a_i,x_{i+1})$ lorsqu'on en connait un $M = (Q,\Sigma^{i+1},\delta)$ pour φ_i. Posons $m_i = \max(|a_1|,\ldots,|a_i|)$. L'idée est la suivante : lorsqu'on lira un symbole $s^j \in \Sigma^i$ (qui correspond à la suite des $j$emes bits $s_1^j,...,s_i^j$ de $a_1,...,a_i$ pour un certain j, on va deviner le bit correspondant de x_{i+1} et obtenir un symbole $(s,s_{i+1}^j) \in \Sigma^{i+1}$. On fait ensuite appel à M pour décider si un des deux choix était concluant. A chaque étape, l'état courant de M' correspondra à un ensemble d'état de M. On précise l'approche ci-dessous.

On pose tout d'abord $Q' = \mathscr{P}(Q) \cup \{\tilde{q}_I\}$ où $\tilde{q}_I \notin Q$ est un nouvel état de départ. Soit Dep, l'ensemble des états de M accessibles à partir de l'état initial de M en ne lisant que des symboles $s_0 = (0,...,0,0)$ et $s_1 = (0,...,0,1)$. On définit la fonction de transition δ' comme suit :

– $\delta'(\tilde{q}_I,\epsilon) = Dep$ signifiant que l'on se place, avant de lire la moindre lettre dans l'état $Dep \in Q'$. Cette première étape ne sert qu'à prendre en compte des valeurs pour x_{i+1} qui pourraient être plus grandes que a_1,..., a_i (ce qui explique que l'on ajoute des '0' non significatifs devant ceux-ci).
– Pour tout $\tilde{q} \in Q' \backslash \{q_I\}$,

$$\delta'(\tilde{q},s) = \{\delta(q,(s,0)) : q \in \tilde{q}\} \cup \{\delta(q,(s,1)) : q \in \tilde{q}\}.$$

– M' accepte si son état final, $\tilde{q}$ contient un état acceptant de M.

Si M' accepte cela signifie qu'un état acceptant, disons q_f de M fait partie de l'état final acceptant de M' et, par définition de δ', qu'il existe une suite de m_i choix $s_{i+1}^1,\ldots,s_{i+1}^m$, un état $q_0 \in Dep$ et m états $q^1,...,q^m = q_f$ tels que :

$$\delta(q_0,s^1 s_{i+1}^1) = q^1,\ldots,\delta(q_0,s^m s_{i+1}^m) = q^m = q_f$$

Cela implique qu'il existe un entier a_{i+1} dont les m derniers bits sont donnés par $s_{i+1}^1,\ldots,s_{i+1}^m$ tels que $\varphi_{i+1}(a_1,...,a_i,x_{i+1}/a_{i+1})$ est vraie dans $\mathbb{N}$. Réciproquement, l'existence d'un entier a_{i+1} tel que $\varphi_{i+1}(a_1,...,a_i,x_{i+1}/a_{i+1})$ est vraie dans $\mathbb{N}$ se traduit par un calcul acceptant de M'. ■

Remarque. La preuve ci-dessus peut être considérablement simplifiée en faisant appel aux résultats élémentaires sur les automates : clôture par union, intersection, complément des langages reconnus par automates (réguliers) et méthodes de déterminisation d'un automate non déterministe.

Chapitre 4

Rudiments de Complexité

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

On a jusqu'à présent parlé de calculabilité au sens large. Après s'être intéressé à la notion de modèles de calcul et avoir défini des notions (convergentes) de fonctions et d'ensembles calculables, nous avons donné un rapide aperçu de quelques problèmes décidables et indécidables. Dans cette partie, on se propose d'aller plus loin sur les notions de décidabilité et de rendre un peu plus précises les mesures de coût ou de ressources algorithmique nécessaires pour décider si une fonction ou un ensemble sont calculables.

4.1 Modèles de calcul

On va privilégier dans la suite de ce chapitre la machine de Turing comme modèle de calcul. Les résultats de simulation vu précédemment permettraient d'utiliser aussi bien des machines à registres, des programmes structurés ou des machines RAMs [1].

Nous allons considérer dans la suite des machines dont tous les calculs s'arrêtent (en acceptant ou en refusant une entrée s'il s'agit d'un problème de décision). Toutefois, beaucoup de notions introduites dans cette partie (arbre de calcul, configuration, etc) restent vraies dans le cas général. On ne revient pas sur le modèle de machine de Turing déterministe et, avant de présenter les modèles de coût, on introduit un nouveau type de machine de Turing.

4.1.1 Machines de Turing non déterministes

Définition 4.1.1 (machine de Turing non déterministe à k rubans) *Une machine de Turing non déterministe est une machine de Turing $M = (Q, \Gamma, \Sigma, \delta, q_0, q_a, q_r)$ dont la fonction de transition est de la forme :*

$$\delta : Q \times \Gamma^k \longrightarrow \mathscr{P}(Q \times \Gamma^k \times \{\lhd, \rhd, -\}^k).$$

A chaque étape, on a donc le choix entre plusieurs transitions possibles. On a considéré dans la définition des machines à plusieurs rubans dont un d'entrée (ce que l'on continuera de faire dans la suite). Informellement, une telle machine accepte une entrée $w \in \Sigma^*$ s'il existe un calcul menant à l'état aceptant et refuse, donc, si tous les calculs mênent à un état refusant (en supposant que tous les calculs s'arrêtent). On formalise cette notion dans la suite.

4.1.2 Arbres de calcul, langages reconnaissables

Configuration d'une machine de Turing. A tout instant d'un calcul, on peut décrire la configuration C_t d'une machine M en donnant son état courant et en décrivant le contenu significatif de chaque

1. Seule la notion de coût d'une simulation d'une machine par une autre devrait être précisée. Nous ne le faisons pas ici faute de temps.

ruban (même si la machine ne s'arrête jamais, à tout instant, la description est finie). Plus précisement, une configuration C sera la donnée d'un uplet $(q, u_1 : v_1, \dots, u_i : v_i, \dots, u_k : v_k)$ où :

- q est l'état courant,
- pour tout $i \leq k$, $u_i \in \Gamma^*$, $v_i \in \Gamma^*$ et $u_i v_i$ est le mot écrit sur le ruban i (il est suivi, sur la droite du ruban par des symboles "blancs"). La tête de lecture se trouve sur la dernière lettre de u_i.

On note $C_{0,\overline{w}}$ la configuration initiale de la machine avec le mot $\overline{w}$ sur le ruban d'entrée. De même, C_a et C_r désignent des configurations acceptantes et rejetantes (i.e. dont l'état q associé est q_a ou q_r).

Si C et D sont deux configurations, on écrira $C \vdash^1_M D$ s'il existe une transition de M permettant de passer de C à D. De même, on écrira $C \vdash^*_M D$ s'il existe une suite de transitions menant de C à D.

Exemple. Soit M une machine à un ruban, pour simplifier, $C = (q, u_1 : v_1))$ une configuration, et $(q', \gamma', \lhd) \in \delta(q, \gamma)$ une des transitions possibles de M. On suppose que la dernière lettre de u_1 est γ i.e. $u_1 = u'_1\gamma$ avec $u'_1 \in \Gamma^*$ et $\gamma \in \Gamma$. Alors, on peut passer de C à la configuration :

$$D = (q', u'_1 : \gamma' v_1),$$

par δ. La tête de lecture étant sur la dernière lettre de u'_1 après la transition.

Définition 4.1.2 *On dit qu'une machine M accepte un mot $\overline{w}$ si $C_{0,\overline{w}} \vdash^*_M C_a$ autrement dit s'il existe une suite de transitions de la machine menant de la configuration initiale à une configuration acceptante.*

Arbre de calcul. On peut décrire le calcul d'une machine M sur une entrée $\overline{w}$ par un arbre : la racine en est la configuration initiale, ses fils sont les configurations obtenues en effectuant toutes les transitions possibles à partir de cette première configuration, puis les configurations obtenues à partir des fils de la première, etc. Dans le cas général, certaines branches peuvent être finies (auquel cas, la configuration feuille a pour état q_r ou q_a) d'autres peuvent être infinies (si le calcul ne termine pas). Si la machine M est déterministe, l'arbre de calcul est réduit à une seule et unique branche quelque soit l'entrée.

Pour faire echo à la définition précédente, une machine M accepte $\overline{w}$ s'il existe un chemin dans l'arbre de calcul menant à une configuration acceptante (i.e. si une feuille "accepte"). Ceci implique que le mot $\overline{w}$ est refusé si toutes les branches de calculs sont soit infinies soit finissent dans un état q_r.

Définition 4.1.3 (langage reconnu par une machine) *On appelle langage reconnu par une machine de Turing M l'ensemble :*

$$L(M) = \{\overline{w} : \ M \textit{ accepte } \overline{w}\}.$$

4.1.3 Fonctions

Les machines décrites jusque là permettent d'accepter ou de refuser des mots. On peut les modifier de façon à leur faire calculer des fonctions. Dans ce cas, on suppose que parmi les rubans de travail, deux d'entre eux ont des utilités particulières : un des rubans est réservé à la lecture de l'entrée ; l'autre, appelé ruban de sortie, est destiné à recevoir le résultat du calcul.

Soit $f : \Sigma^* \to \Sigma^*$ et M une machine de Turing comme ci-dessus *déterministe*. On dit que M calcule la fonction T si, pour toute entrée $\overline{w} \in \Sigma^*$, la machine s'arrête dans l'état acceptant (après un certain nombre d'étapes) avec le mot $f(w)$ écrit sur son ruban de sortie.

La définition de machine calculant une fonction peut s'étendre au cas non déterministe.

4.2 Mesures de complexité en temps et en espace

On appelle longueur d'un mot $\overline{w} = w_1 \dots w_n \in \Sigma^*$, que l'on note $|\overline{w}|$, le nombre de symboles de Σ le composant. Ici, $|\overline{w}| = n$.

Soit M est une machine de Turing sur l'alphabet (d'entrée) Σ. On appelle $t_M(\overline{w})$ le nombre d'étapes requis jusqu'à ce que tous les calculs de M (si M est non déterministe) s'arrêtent sur l'entrée $\overline{w}$ (on poserait $t_M(\overline{w}) = \infty$ si M ne s'arrêtait pas sur $\overline{w}$). De même, on appelle $s_M(\overline{w})$ l'espace (i.e. le plus grand indice d'une case de ruban de travail[2] visité par la tête de lecture) utilisé par M. On peut mesurer les ressources utilisées par une machine M de la façon suivante

Définition 4.2.1 (complexité dans le pire des cas) *Soit T_M et S_M les deux fonctions de $\mathbb{N}$ dans $\mathbb{N} \cup \{\infty\}$ définies par :*

$$T_M(n) = max\{t_M(\overline{w}) : \overline{w} \in \Sigma^* \text{ et } |\overline{w}| = n\},$$

$$S_M(n) = max\{s_M(\overline{w}) : \overline{w} \in \Sigma^* \text{ et } |\overline{w}| = n\}.$$

T_M décrit la complexité en temps de la machine M et S_M sa complexité en espace.

Chacune de ses fonctions renvoie, pour toute longueur d'entrée n, respectivement le temps maximal et l'espace maximal utilisée par un calcul de M sur une entrée de taille n.

4.2.1 Classes de complexité en temps et en espace

On peut maintenant définir les mesures principales de complexité en temps et en espace.

Définition 4.2.2 (classes en temps) *Soit $T : \mathbb{N} \to \mathbb{N}$. Un langage $L \subseteq \Sigma^*$ est dans la classe* $\mathrm{DTIME}(T(n))$ *(resp.* $\mathrm{NTIME}(T(n))$*) s'il existe une machine de Turing M déterministe (resp. non déterministe) à plusieurs rubans telle que :*
- *$L = L(M)$,*
- *$T_M(n) = O(T(n))$ i.e. il existe une constante c telle que, pour tout entier[3] n, tout calcul de M sur une entrée de taille n s'arrête en un temps borné par $cT(n)$.*

On donne une définition similaire pour les classes de complexité en espace.

Définition 4.2.3 (classes en espace) *Soit $S : \mathbb{N} \to \mathbb{N}$. Un langage $L \subseteq \Sigma^*$ est dans la classe* $\mathrm{DSPACE}(S(n))$ *(resp.* $\mathrm{NSPACE}(S(n))$*) s'il existe une machine de Turing M déterministe (resp. non déterministe) à plusieurs rubans telle que :*
- *$L = L(M)$,*
- *$S_M(n) = O(S(n))$ i.e. il existe une constante c telle que, pour tout entier n, tout calcul de M sur une entrée de taille n utilise un espace borné par $cS(n)$ (sur ses rubans de travail).*

Ces deux définitions nous permettent d'introduire des notations pour les mesures de complexité les plus usuelles, à savoir celles concernants le temps et l'espace polynomial.

Définition 4.2.4 *On définit :*

$$\mathrm{P} = \bigcup_{d \geq 1} \mathrm{DTIME}(n^d),\ \mathrm{NP} = \bigcup_{d \geq 1} \mathrm{NTIME}(n^d),$$

$$\mathrm{PSPACE} = \bigcup_{d \geq 1} \mathrm{DSPACE}(n^d),\ \mathrm{NPSPACE} = \bigcup_{d \geq 1} \mathrm{NSPACE}(n^d),$$

$$\mathrm{L} = \mathrm{DSPACE}(\log_2(n)),\ \mathrm{NL} = \mathrm{NSPACE}(\log_2(n))\ [4],$$

$$\mathrm{E} = \mathrm{DTIME}(2^{O(n)}),\ \mathrm{NE} = \mathrm{NTIME}(2^{O(n)}).$$

2. Le ruban d'entrée n'est jamais modifié, on évalue la complexité sur l'espace réellement utilisé pour le travail de la machine.
3. on peut excepter les petites valeurs de n
4. Ces deux classes se notent aussi parfois respectivement LOGSPACE et NLOGSPACE.

4.3 Inclusions et hiérarchies entre les classes

4.3.1 Fonctions constructible en temps et espace

De façon surprenante, on ne peut pas considérer n'importe quelle borne pour mesurer la complexité. Le résultat suivant montre que certaines fonctions amènent à des situations contre-intuitive.

Théorème 4.3.1 ("Gap Theorem") *Il existe une fonction récursive $T : \mathbb{N} \longrightarrow \mathbb{N}$ telle que :*

$$\text{DTIME}(T(n)) = \text{DTIME}(2^{T(n)})$$

On utilise généralement comme bornes de fonction de complexité, des fonctions f que l'on peut "construire". L'idée intuitive est que l'on doit être en mesure de construire une machine fonctionnant comme une horloge pour la fonciton f i.e. s'arrêtant à l'instant où, pour chaque longueur d'entrée n, on atteint la valeur de $f(n)$.

Définition 4.3.2 *Une fonction $f : \mathbb{N} \longrightarrow \mathbb{N}$ avec $f(n) \geq n$, pour tout $n \in \mathbb{N}$ est constructible en temps s'il existe une machine de Turing M déterministe qui, sur l'entrée 1^n (constituée de n symboles 1 consécutifs) calcule la représentation binaire de $f(n)$ en temps $O(f(n))$. Une fonction $f : \mathbb{N} \longrightarrow \mathbb{N}$ avec $f(n) \geq$ « n, pour tout $n \in \mathbb{N}$ est constructible en espace s'il existe une machine de Turing M déterministe qui, pour sur l'entrée 1^n, calcule la représentation binaire de $f(n)$ en espace $O(S(n))$.*

La plupart des fonctions usuelles : polynomiales, exponentielles, etc sont constructibles en temps. La fonction log ou les fonctions racines sont aussi constructibles en espace.

4.3.2 Relations d'inclusion entre les classes

La preuve de la première proposition ci-dessous est immédiate en remarquant que toute machine déterministe est une machine non déterministe (avec un seul choix).

Proposition 4.3.1 *Pour toute fonction $T : \mathbb{N} \to \mathbb{N}$:*

$$\text{DTIME}(T(n)) \subseteq \text{NTIME}(T(n)) \textit{ et } \text{DSPACE}(T(n)) \subseteq \text{NSPACE}(T(n)).$$

Démonstration. Immédiat car toute machine déterministe est une machine non déterministe (sans choix).

Proposition 4.3.2 *Pour toute fonction $T : \mathbb{N} \to \mathbb{N}$:*

$$\text{NTIME}(T(n)) \subseteq \text{DSPACE}(T(n)).$$

Démonstration (idée). On voit facilement que toute machine non déterministe peut être normalisée en une machine ou à chaque étape exactement deux choix sont possibles[5]. Soit M une machine de Turing non déterministe fonctionnant en temps polynomial $p(n)$ et $L(M)$ le langage reconnu par M. On peut simuler M par une machine M' avec un ruban supplémentaire (par exemple) de la façon suivante :

- M' écrit successivement tous les mots $y \in \{1,2\}^{p(n)}$ chacun symbolisant une liste de $p(n)$ choix non déterministes possibles. - Pour chacun des mots y, M' simule M en prenant à chaque étape i, le premier choix de transition si $y_i = 1$ et le deuxième choix sinon. M' accepte si le calcul termine dans l'état q_a et passe au y suivant sinon.

Clairement, la première étape requiert un espace $p(n)$ et la seconde, fonctionnant en temps $p(n)$ utilise donc aussi un espace de cet ordre. ■

Proposition 4.3.3 *Pour toute fonction $T : \mathbb{N} \to \mathbb{N}$ telle que $\log_2 n \leq T(n))$:*

$$\text{NSPACE}(T(n)) \subseteq \bigcup_{c \geq 1} \text{DTIME}(c^{T(n)}).$$

5. Le faire en exercice

Remarque. Pour aborder l'inclusion (plus faible) DSPACE($T(n)$) ⊆ DTIME($2^{O(n)}$), on peut montrer que : si M est une machine de Turing fonctionnant sur toute entrée de taille n en espace $cT(n)$ (pour une certaine constante c), alors il existe M' telle que $L(M) = L(M')$ et M' s'arrête sur toute entrée de taille n en un temps borné par $d^{T(n)}$ (pour un certain d, là aussi).

L'argument principal pour montrer ce résultat vient du fait que toute machine fonctionnant en espace $cT(n)$ a un nombre de configurations différentes borné par un certain $d^{T(n)}$ et que toute execution plus longue que $d^{T(n)}$ voit se répéter deux fois (au moins) une même configuration : ce qui implique, pour toute machine déterministe, que celle ci est en train de boucler et ne s'arrêtera pas. Il est donc justifié de l'arrêter après $d^{T(n)}$ dans un état refusant.

Pour l'inclusion NSPACE($T(n)$) ⊆ DTIME($2^{O(n)}$), on généralise l'approche en construisant un *graphe de configuration.*

On obtient comme corollaire :

Corollaire 4.3.3 *La suite d'inclusion suivante est vraie :*

$$\mathrm{L} \subseteq \mathrm{NL} \subseteq P \subseteq NP \subseteq \mathrm{PSPACE} \subseteq \mathrm{NPSPACE} \subseteq \mathrm{E}.$$

4.3.3 Hiérarchies en temps et espace

Dans cette partie, on va énoncer un certain nombre de résultats de séparation classiques sur les classes de complexité en temps et en espace. On ne prouvera pas ces résultats dans le corus (faute de temps) qui sont donnés ici à des fins bibliographiques. Le principe général est que : si on a un peu plus d'espace (resp. de temps) alors on peut décider strictement plus de langage.

On a besoin au préalable de quelques résultats sur les machines universelles.

Soit Σ un alphabet. A toute machine M on peut associer un code $\langle M\rangle$ dans l'alphabet Σ (dans un sens que l'on précisera). De même, pour tout mot w sur un alphabet Σ', on peut associer un mot $\langle w\rangle$ sur Σ qui le code (avec la contrainte que si $w \neq w'$ alors $\langle w\rangle \neq \langle w'\rangle$). Bien entendu, à toute lettre de Σ' correspond un mot de Σ si $|\Sigma'| > |\Sigma|$.

On dit que U est une machine de Turing universelle si, pour toute entrée $\langle M\rangle : \langle w\rangle$ on a :

$$U \text{ accepte } \langle M\rangle : \langle w\rangle \text{ ssi } M \text{ accepte } w.$$

Cette notion usuelle en calculabilité va nous servir (un peu raffinée) pour la séparation des classes de complexité.

Théorème 4.3.4 *Pour tout k, il existe une machine de Turing à k rubans U telle que sur toute entrée $\langle M\rangle : \langle w\rangle$ où M est une machine à k rubans : U accepte $\langle M\rangle : \langle w\rangle$ ssi M accepte w.*

Soit $l_M = |\langle M\rangle|$. Si M utilise un espace $O(T(n))$ (sur tout w, t.q. $|w| = n$) alors U utilise un espace $O(l_M T(n))$. Si M utilise un temps $O(T(n))$ alors U utilise un temps $O(l_M^2 T(n))$.

Remarque. La preuve donne la simulation d'une machine à k rubans par une machine universelle à $k+1$ ruban. Moyennant un effort supplémentaire (considérer un (des) ruban(s) comme ayant plusieurs pistes et "trainer" le code de $\langle M\rangle$ avec soi tout au long du calcul (ce qui rajoute un facteur multiplicatif l_M pour simuler chaque étape)), on peut obtenir la preuve du résultat général.

Théorème 4.3.5 (hiérarchie en espace) *Pour toute fonction $S_2 : \mathbb{N} \to \mathbb{N}$ constructible et toute fonction $S_1 : \mathbb{N} \to \mathbb{N}$ telle que $S_1(n) \geq \log n$ et $S_1(n) = o(S_2(n))$, la classe* DSPACE($S_1(n)$) *est strictement incluse dans* DSPACE($S_2(n)$).

Le théorème de hiérarchie pour le temps que l'on peut obtenir est un peu moins "performant".

Théorème 4.3.6 (hiérarchie en temps) *Pour toute fonction $T_2 : \mathbb{N} \to \mathbb{N}$ constructible et toute fonction $T_1 : \mathbb{N} \to \mathbb{N}$ telle que $T_1(n) \geq n$ et $T_1(n)\log n = o(T_2(n))$, la classe* DTIME($T_1(n)$) *est strictement incluse dans* DTIME($T_2(n)$).

On obtient, entre autres comme corollaire de ces deux résultats, une hiérarchie stricte à l'intérieur de P et PSPACE basée sur le degré du polynome :

Corollaire 4.3.7 *Soient r, s deux entiers tels que* $1 \le r < s$*, alors :*

$$\mathrm{DTIME}(n^r) \subset \mathrm{DTIME}(n^s) \textit{ et } \mathrm{DSPACE}(n^r) \subset \mathrm{DSPACE}(n^s).$$

De même :

$$\mathrm{P} \subset \mathrm{E} \textit{ et } \mathrm{NP} \subset \mathrm{NE}.$$

Les résultats précédents impliquent aussi que, même si on ne sait pas si P $\neq$ NP on sait que soit P $\neq$ NP soit NP $\neq$ E.

Pour ce qui concerne les classes non déterministes en temps, on a le même genre de résultat mais plus fin.

Théorème 4.3.8 (hiérarchie non déterministe en temps) *Pour toutes fonctions $T_1 : \mathbb{N} \to \mathbb{N}$ et $T_2 : \mathbb{N} \to \mathbb{N}$ constructibles telles que $T_1(n+1) = o(T_2(n))$, la classe* $\mathrm{NTIME}(T_1(n))$ *est strictement incluse dans* $\mathrm{NTIME}(T_2(n))$.

Il existe de même une hiérarchie stricte pour les classes en espace non déterministe (mais on prouvera bien plus sur l'espace non déterministe plus loin dans le cours).

4.4 Définition alternative de NP

Dans toute cette partie, $\Sigma, \Sigma_1, \Sigma_2, \Sigma_3$ sont des alphabets quelconques. On va donner une définition alternative de NP (une sorte de forme normale) où en quelque sorte le recours au non déterminisme se fait en début d'algorithme.

Définition 4.4.1 *Soit $A \subseteq \Sigma^*$, $A \in$ NP s'il existe un prédicat binaire $B \subseteq \Sigma^* \times \Sigma^*$ dans* P *et une polynôme p tels que :*

$$A = \{x : \exists y\ |y| \le p(|x|) \textit{ et } B(x, y)\}.$$

Cette définition présente la classe NP comme une projection de la classe P.

Voir que les deux approches alternatives de NP sont équivalentes n'est pas très dur. Si A est reconnu par une Machine de Turing non déterministe en temps polynomial alors pour toute instance x, $x \in A$ ssi il existe une suite de choix y (de taille bornée par le nombre d'étapes, donc polynomial) pour la fonction de transition menant à un état acceptant. Le prédicat B prenant x et y en entrée et acceptant (x, y) si y est bien une telle séquence est décidable en temps polynomial.

Réciproquement, supposons $A = \{x : \exists y\ |y| \le p(|x|)$ et $B(x, y)\}$ avec $B \in P$. La machine de Turing non déterministe M avec $|\Sigma|$ choix de transitions à chaque étape qui écrit au fur et à mesure la suite y de ses choix pendant $p(|x|)$ étapes et lance B sur x et le y ainsi obtenue reconnaît bien A en temps polynomial.

L'intérêt de l'approche prédicative de NP est qu'elle propose une vision simplissime des algorithmes non déterministes. Un problème A est dans NP si pour toute instance x dans A, il existe un témoin y (de taille polynomiale en la taille de x) de l'appartenance de x à A. La vérification de ce témoin se faisant en temps polynomial.

Illustrons cette dernière approche avec le problème suivant.

SAT
Entrée : Une formule propositionnelle φ sous forme normale conjonctive
Question : φ est-elle satisfaisable ?

Soit φ une formule propositionnelle. $\varphi \in$ SAT s'il existe une assignation de valeur de vérités $I : V \longrightarrow \{0, 1\}$ pour les variables V de φ, tel que I satisfait φ. L'assignation I est de taille (sous) linéaire en la taille de φ : on peut le voir, par exemple, comme un mot $\{0, 1\}^{|V|}$. De plus la vérification se fait facilement en temps polynomial après substitution de $I(p)$ à p, pour tout $p \in V$, dans φ. Donc SAT $\in$ NP.

4.5 Réductions entre problèmes

Si $A \subseteq \Sigma^*$, on notera $A(x)$ l'appartenance de x à A.

Définition 4.5.1 *Soient $A \subseteq \Sigma_1^*$ et $B \subseteq \Sigma_2^*$, on dit que A se réduit à B s'il existe une fonction $f : \Sigma_1^* \to \Sigma_2^*$ telle que, pour toute entrée $w \in \Sigma_1^*$:*

$$A(x) \text{ ssi } B(f(x))$$

Lorsque f est calculable en temps polynomial, on parlera de réduction polynomiale de A vers B [6] et on notera $A \leq_P B$.

Si f est calculable en espace logarithmique, on parlera de réduction logarithmique (ou dans LOGSPACE*) de A vers B et on notera $A \leq_L B$.*

La réduction logarithmique est a priori moins forte que la réduction polynomiale : comme LOGSPACE $\subseteq$ P lorsque $A \leq_L B$ on a aussi $A \leq_P B$. Dans les faits, pourtant, à chaque fois que l'on connait une réduction polynomiale elle s'avère être logarithmique (modulo quelques modifications parfois).

Définition 4.5.2 (problèmes NP-complets) *Un problème A est* NP*-complet (par réduction polynomiale) si :*

- *$A \in$ NP et*
- *Pour tout $B \in$ NP, $B \leq_P A$*

Lorsqu'on a seulement la deuxième condition, A est dit NP*-dur ou* NP*-difficile.*

De même, pour le temps polynomial, on peut définir une notion de “difficulté”.

Définition 4.5.3 (problèmes P-complets) *Un problème A est* P*-complet (par réduction dans* LOGSPACE*) si :*

- *$A \in$ P et*
- *Pour tout $B \in$ P, $B \leq_L A$*

Lorsqu'on a seulement la deuxième condition, A est dit P*-dur.*

On peut parler de problèmes complets pour à peu près n'importe quelle classe de complexité. Le vrai critère est alors la réduction choisie : elle ne doit pas être trop faible (au risque de ne “capturer” que trop peu de problèmes) ni trop forte (au risque "d'écraser" tous les problèmes les uns sur les autres [7]).

On vérifie facilement le résultat suivant.

Proposition 4.5.1 *Soient $f : \Sigma_1^* \to \Sigma_2^*$ et $g : \Sigma_2^* \to \Sigma_3^*$ calculables en temps polynomial (resp. en espace logarithmique) alors $g \circ f$ est calculable en temps polynomial (resp. en espace logarithmique).*

Démonstration (idée). On donne quelques arguments pour la clôture par composition dans le cas logarithmique. On se rappelle que pour les classes en espace, la machine de Turing possède un ruban d'entrée (en lecture seule), un ruban de sortie (en écriture seule) et un ruban de travail (sur lequel l'espace de travail utilisé est mesuré). Pour une entrée x donnée, la fonction g ne peut pas travailler directement sur $f(x)$: écrire $f(x)$ peut nécessiter plus qu'un espace logarithmique. Soit M_g et M_f les machines calculant f et g. La machine M calculant $g \circ f$ va simuler M_g sur $f(x)$ de façon indirecte : elle prend x en entrée et garde la trace (sous forme d'un compteur c) de la position à tout instant de la tête de lecture de M_g sur $f(x)$. A chaque transition de M_g, elle inspecte c et lance la simulation de M_f sur x jusqu'à ce que celle-ci produise le c^{eme} bit de $f(x)$ (elle efface ceux d'avant mais garde trace du nombre d de bits de la sortie produits) Une fois qu'elle a celui-ci, la transition de M_g peut être effectuée, la tête de lecture se retrouve dans une nouvelle position, le compteur c est modifié en conséquence et la manipulation décrite plus haut recommence. Les indices c et d que M doit garder en mémoire sont de l'ordre de $\log|f(x)|$. Or comme M_f est dans LOGSPACE, le nombre d'étapes maximales (et donc $|f(x)|$) est polynomial en $|x|$ donc $\log|f(x)| = O(\log|x|)$.

On obtient comme corollaire la transitivité des réductions.

6. Sans plus de précision pour l'instant. La réduction présentée ici s'appelle "many-one" dans la littérature
7. Quel sens aurait une théorie de la P-complétude par des réductions dans P ?

Corollaire 4.5.4 *Les réductions $\leq_{\mathrm{P}}$ et $\leq_{\mathrm{L}}$ sont transitives i.e. si $A \leq_{\mathrm{P}} B$ et $B \leq_{\mathrm{P}} C$ alors $A \leq_{\mathrm{P}} C$ (de même pour $\leq_{\mathrm{L}}$).*

L'avantage de considérer des réductions est qu'elles évitent une étude complète de la complexité des problèmes considérés. "Réduire" c'est en quelque sorte "transférer la complexité".

Proposition 4.5.2 *Soient A et B dans Σ^*, les propriétés suivantes sont vraies.*

- *Si $A \leq_{\mathrm{P}} B$ et $B \in \mathrm{P}$ alors $A \in \mathrm{P}$.*
- *Si $A \leq_{\mathrm{L}} B$ et $B \in \mathrm{LOGSPACE}$ alors $A \in \mathrm{LOGSPACE}$.*
- *Si $A \leq_{\mathrm{P}} B$ et A est NP-dur alors B est NP-dur.*
- *Si $A \leq_{\mathrm{L}} B$ et A est P-dur alors B est P-dur.*

Démonstration. Pour le premier point, soit x une instance de A, M_f l'algorithme polynomial calculant la réduction f et M_B la machine décidant M (dans P aussi). On compose les algorithmes : M_f transforme x en $f(x)$ puis on lance M_B sur $f(x)$. La compossition est bien un processus se déroulant en temps polynomial. Le deuxième point est analogue.

Pour le troisième et quatrième point, on applique la transitivité de $\leq_{\mathrm{P}}$ et $\leq_{\mathrm{L}}$.

4.6 NP-complétude de SAT

Pour pouvoir démarrer une théorie de la NP-complétude, il faut au moins avoir un premier problème NP-complet. C'est le cas du problème SAT.

SAT
Entrée : Une formule propositionnelle φ sous forme normale conjonctive
Question : φ est-elle satisfaisable ?

Théorème 4.6.1 (Cook'71, Levin'69) SAT *est* NP*-complet*

Démonstration. L'appartenance de SAT à NP a déjà été vue. On va donner une réduction qui code le calcul d'une machine N.D. polynomiale sur une entrée w par une formule propositionnelle. Plus précisement soit $M = (Q, \Gamma, \Sigma, \delta, q_0, q_a, q_r)$ une machine de Turing N.D. à un ruban telle que $L(M) \in \mathrm{NP}$ et $w \in \Sigma^*$, on construit une formule propositionnelle $\Phi_{M,w}$ (de taille polynomiale en $|w|$) telle que :

$$M \text{ accepte } w \text{ ssi } \Phi_{M,w} \text{ est satisfaisable.}$$

Avant de commencer, on remarque que si $L(M) \in \mathrm{NP}$ alors il existe un polynôme P telle que M s'arrête sur toute entrée w en un temps borné par $P(|w|)$ et un espace lui aussi borné par $P(|w|)$. Dans la suite, on pose $w = w_1 \dots w_n$ et donc $|w| = n$, $Q = \{0, \dots, |Q|-1\}$, $\Sigma = \{0, \dots, |\Sigma|-1\}$, $\Gamma = \{0, \dots, |\Gamma|-1\}$. De même, les mouvements de la machine de Turing sont pris dans l'ensemble $\{+1, -1, 0\}$. On renomme aussi q_a en q_1 et q_r en q_2. Enfin, on suppose que la fonction de transition de la machine de Turing est totale : en dehors des état finaux, une transition est prévue quelque soit l'état et le symbole lus.

La formule $\Phi_{M,w}$ que l'on construit est la conjonction de quatre formules :

$$\textsc{Init}, \textsc{Contrainte}, \textsc{Transition}_M, \textsc{Final}.$$

Les formules Init et Final vont coder respectivement la configuration initiale de la machine et sa configuration finale ; Contrainte est un ensemble de clauses qui doit être satisfaite par toute machine et $\textsc{Transition}_M$ code les transitions de la machine M.

Dans la suite, on va introduire des variables propositionnelles indicées (par le temps, les numéros de case de ruban, etc). Plutôt que de les noter sous la forme, par exemple, $q_{i,t}$, on les notera sous une forme "relationnelle" $q(i,t)$.

Les conditions initiales La formule INIT va être la conjonction des clauses suivantes.

- ⋆ I_1 : $\text{Case}(0,i,\sigma)$ avec $\sigma = w_i$ si $i \leq n$ et $\sqcup$ si $n < i \leq P(n)$.
  Ces clauses signifient que la case i du ruban contient le symbole σ à l'instant 0. Il y a $P(n)$ telles clauses.
- ⋆ I_2 : $\text{Etat}(0,0)$
  A l'instant 0, on est dans l'état initial q_0.
- ⋆ I_3 : $\text{Tete}(0,1)$
  A l'instant 0, la tête de lecture est sur la case 1.

Les contraintes de machines Cette partie est nécessaire (bien que peu informative) : on code le fait que les variables propositionnelles vont bien imiter le fonctionnement d'une machine de Turing. La formule CONTRAINTE va être la conjonction des clauses suivantes.

- ⋆ C_1 : $\text{Case}(t,i,1) \vee \cdots \vee \text{Case}(t,i,|\Gamma|)$
  A chaque instant t, chaque case i contient un symbole de Γ (possiblement $\sqcup$ ou dans $\Sigma \subseteq \Gamma$). On doit construire $P(n)^2$ telle clause : $1 \leq i \leq P(n)$ et $1 \leq t \leq P(n)$
- ⋆ C_2 : $\text{Case}(t,i,\gamma) \rightarrow \neg\text{Case}(t,i,\gamma')$ pour $\gamma \neq \gamma'$
  A chaque instant t, la case i ne peut contenir qu'un seul symbole γ. On a $O(P(n)^2)$ clauses.
- ⋆ C_3 : $\text{Etat}(t,1) \vee \cdots \vee \text{Etat}(t,|Q|)$
  A chaque instant t, la machine est dans au moins un état q. Il y a $O(P(n))$ clauses.
- ⋆ C_4 : $\text{Etat}(t,q) \rightarrow \neg\text{Etat}(t,q')$ pour $q \neq q'$.
  A chaque instant t, la machine ne peut être que dans un seul état q. On a $O(P(n))$ clauses.
- ⋆ C_5 : $\text{Tete}(t,1) \vee \cdots \vee \text{Tete}(t,P(n))$
  A chaque instant t, La tête de lecture visite au moins une case. Il y a $O(P(n))$ clauses.
- ⋆ C_6 : $\text{Tete}(t,i) \rightarrow \neg\text{Tete}(t,i')$ pour $i \neq i'$.
  A chaque instant t, la tête de lecture ne peut visiter qu'au plus une case i. On a $O(P(n)^2)$ clauses.
- ⋆ C_7 : $\text{Tete}(t,i) \wedge \text{Case}(t,i',\gamma) \rightarrow \text{Case}(t+1,i',\gamma)$ pour $i \neq i'$
  Si la tête de lecture est sur la case i à l'instant t, le contenu des autres cases $i' \neq i$ ne sera pas modifié à l'instant d'après. Il y a $O(P(n)^3)$ clauses comme celle-ci.

Les transitions de la machine Voici la partie qui est réellement dépendante de la machine M de départ. On va coder par des formules les transitions de la machine. On rappelle que pour une machine non déterministe la fonction de transition δ est de la forme :

$$\delta : Q \times \Gamma \longrightarrow \mathscr{P}(Q \times \Gamma \times \{+1,-1,0\}).$$

Chaque $\delta(q,\gamma)$ est donc un ensemble contenant des triplets de la forme (q',γ',d). On construit TRANSITION_M la conjontion de clauses du même type suivant.

$$T_1 : \ \text{Tete}(t,i) \wedge \ \text{Etat}(t,q) \wedge \text{Case}(t,i,\gamma) \longrightarrow \bigvee_{(q',\gamma',d)\in\delta(q,\gamma)} \text{Tete}(t+1,i+d) \wedge \text{Etat}(t+1,q') \wedge \text{Case}(t+1,i,\gamma').$$

La signification en est [8] : si à un instant t, la tête de lecture est sur la case i que l'on lit le symbole γ sur cette case et que l'on est dans l'état q alors, à l'instant suivant $t+1$, on a pour conséquence sur l'état, le symbole écrit et la position de la tête le résultat d'une des transitions possibles (q',γ',d) à partir de (q,γ).

Là encore, on a besoin de $O(P(n)^2)$ telles clauses.

Remarque. On aurait pu procéder autrement. Notons k le nombre maximal de choix possibles pour chaque transition i.e.

8. formellement, on devrait rajouter une formule disant qu'une fois que l'état acceptant q_1 ou l'état rejetant q_2 sont atteints il ne se passe plus aucun changement (cela pourrait être pratique dans d'autres contextes) donc (pour $e = 1,2$) : $\text{Tete}(t,i) \wedge \text{Etat}(t,e) \wedge \text{Case}(t,i,\gamma) \longrightarrow \text{Tete}(t+1,i) \wedge \text{Etat}(t+1,e) \wedge \text{Case}(t+1,i,\gamma)$

$$\delta(q,\gamma) = \{(q_1,\gamma_1,d_1),\dots,(q_k,\gamma_k,d_k)\}$$

On introduit une nouvelle série de variable Choix(t,c) pour $c \in \{1,\dots,k\}$ et $t \leq P(n)$. Cette variable indique que le choix effectué parmi les transitions possibles à l'instant t est le numéro c. Dans ce cas, on doit construire une nouvelle formule TRANSITION'$_M$ obtenu par conjonction de formules du type :

$$\begin{aligned} T_1' :\ & \text{Tete}(t,i) \wedge \text{Etat}(t,q) \wedge \text{Case}(t,i,\gamma) \wedge Choix(t,c) \longrightarrow \text{Tete}(t+1,i+d_c) \\ & \text{Tete}(t,i) \wedge \text{Etat}(t,q) \wedge \text{Case}(t,i,\gamma) \wedge Choix(t,c) \longrightarrow \text{Etat}(t+1,q_c) \\ & \text{Tete}(t,i) \wedge \text{Etat}(t,q) \wedge \text{Case}(t,i,\gamma) \wedge Choix(t,c) \longrightarrow \text{Case}(t+1,i,\gamma_c) \end{aligned}$$

Il faut rajouter les formules indiquant qu'à l'instant t, un seul choix est effectué :

$$T_2' : (\bigvee_{c=1}^{k} Choix(t,c)) \wedge (\bigwedge_{c \neq c' \in \{1,\dots,k\}} \neg Choix(t,c) \vee \neg Choix(t,c')).$$

La configuration finale Il nous reste a décrire le fait qu'à l'instant $P(n)$ la machine est dans un état acceptant. On écrit pour cela la formule FINAL (on rappelle que l'état final acceptant est noté q_1) :

$$\text{FINAL} \equiv \text{Etat}(P(n),1)$$

.

Si M accepte w en temps borné par $P(n)$, on voit facilement que $\Phi_{M,w}$ est satisfaisable : la formule imite le comportement de la machine avec les mêmes conditions initiales. De même, si $\Phi_{M,w}$ est satisfaisable, les contraintes imposées font que l'assignation code le calcul acceptant d'une machine de Turing fonctionnant en temps au plus $P(n)$. On le montre formellement par induction.

Les clauses, en nombre polynomial, sont construites de façon très uniforme. Cette construction peut être décrite par un algorithme fonctionnant en espace logarithmique : pour chaque type de clause, on a besoin de se souvenir simplement des indices (t, i, i' parfois) qui occupent un espace logarithmique.

Soit $k \in \mathbb{N}^*$. Une k-clause est une clause comportant au plus k littéraux. Soit le problème de satisfaisabilité suivant.

k-SAT
Entrée : Une formule propositionnelle φ constituée de k-clauses
Question : φ est-elle satisfaisable ?

Proposition 4.6.1 *Pour tout $k \geq 3$, k-SAT est NP-complet.*

Démonstration. Pour $k = 3$ (suffisant). L'appartenance à NP dérive de celle de SAT. La NP-complétude s'obtient par réduction à partir de SAT. On peut par exemple réécrire toute clause $C \equiv p_0 \vee p_1 \vee \cdots \vee p_n$ (avec $n > 3$ en la conjonction des clauses suivantes : $(p_0 \vee p_1 \leftrightarrow q_1)$, $(q_1 \vee p_2 \leftrightarrow q_2)$,…, $(q_{n-1} \vee p_n \leftrightarrow q_n)$ et q_n où $q_1,\dots,q_n$ sont de nouvelles variables. Chaque clause $(q_{i-1} \vee p_i \leftrightarrow q_i)$ peut ensuite se réécrire comme la conjonction de $(q_i \vee \neg q_{i-1})$, $(q_i \vee \neg p_i)$, et $(\neg q_i \vee q_{i-1} \vee p_i)$.

On remarque assez facilement qu'exiger que les clauses soient de longueurs *exactement* k aurait donné le même résultat.

4.7 Quelques exemples de problèmes NP-complets

Des milliers de problèmes ont été prouvés NP-complets dans beaucoup de domaine de l'informatique, des mathématiques, de l'écomomie, des systèmes industriels, etc. On en cite quelques uns parmi les plus connus et les plus anciens (voir [**?**] pour une liste très importante et très diverse de tels problèmes).

VERTEX-COVER (recouvrement par sommets)
Entrée : un graphe $G = \langle V, E\rangle$ et un entier k.
Question : existe-t-il un recouvrement par sommets de taille au plus k dans G i.e. $V' \subseteq V$ t.q. $|V'| = k$ et $\forall e = (x, y) \in E,\ x \in V'$ ou $y \in V'$?

CLIQUE (sous-graphe complet)
Entrée : Un graphe $G = \langle V, E\rangle$ et un entier k.
Question : existe-t-il un sous-graphe complet de taille k dans G i.e. $V' \subseteq V$ t.q. $|V'| = k$ et $\forall x \in V' \forall y \in V',\ x \neq y \rightarrow E(x, y)$?

CIRCUIT HAMILTONIEN
Entrée : un graphe $G = \langle V, E\rangle$.
Question : G contient-t-il un circuit hamiltonien i.e. existe-t-il un ordre sur les sommets $v_1,\ldots,v_n$ tel que $E(v_i, v_{i+1})$ pour tout $i \leq n$ et $E(v_n, v_1)$?

PARTITION
Entrée : un ensemble fini A et une fonction $taille : A \mapsto \mathbb{Z}^+$.
Question : existe-t-il un $A' \subseteq A$ t.q. $\sum_{a\in A'} taille(a) = \sum_{a\in A-A'} taille(a)$

k-COLORIAGE
Entrée : un graphe $G = \langle V, E\rangle$.
Question : les sommets de G peuvent-ils être coloriés avec k couleurs de telle sorte que deux sommets adjacents dans G ne portent pas la même couleur ?

Ce problème est déjà NP-complet pour $k = 3$ est en fait un cas particulier d'un problème du type suivant.

HOMOMORPHISME DE GRAPHE (appelé aussi : H-coloriage)
Entrée : un graphe $G = \langle V, E\rangle$ et un graphe $H = \langle V', E'\rangle$.
Question : existe-t-il un homomorphisme de G vers H i.e. une application $h : V \mapsto V'$ telle que $\forall x \in V \forall y \in V,\ E(x, y) \rightarrow E(f(x), f(y))$?

Ce problème est NP-complet en général même si H ne fait pas partie de l'entrée (i.e. est fixé). Le 3-coloriage est un problème d'homomorphisme vers le triangle.

4.8 Des variantes plus faciles de la satisfaisabilité

Une formule est sous forme normale disjonctive (DNF) si elle est écrite comme disjonction de clauses conjonctives i.e. sous la forme $\bigvee_{i=1}^{n} p_i^1 \wedge \ldots \wedge p_i^{k_i}$. On peut définir une première variante de SAT.

DNF-SAT
Entrée : Une formule propositionnelle φ sous forme DNF
Question : φ est-elle satisfaisable ?

Proposition 4.8.1 *le problème* DNF-SAT *est dans* P.

Démonstration. Pour qu'une formule *DNF* φ soit satisfaite il faut et il suffit qu'une seule de ses clauses le soit. L'algorithme revient donc à tester s'il y a une clause non contradictoire i.e. une clause C où, pour toute variable p, p et $\neg p$ n'apparaissent pas simultanement. ■

Remarque. On sait que DNF-SAT ∈ P, CNF-SAT est NP-complet et que l'on peut toujours trouver une forme *CNF* à partir d'une forme *DNF* et réciproquement. Cela pose-t-il problème ? pourquoi ?

Une clause disjonctive est dite sous forme de *Horn* si elle contient au plus un littéral positif, autrement dit si elle peut s'écrire sous la forme :

$$p_1 \wedge \cdots \wedge p_k \to q,$$

où $p_1, \ldots, p_k, q$ sont des littéraux. Une formule propositionnelle sous forme clausale sera dite de Horn si toutes ses clauses sont de Horn. On définit le problème suivant autour de SAT.

HORN SAT
Entrée : Une formule propositionnelle φ de Horn
Question : φ est-elle satisfaisable ?

On peut montrer que ce problème est aussi plus facile que le cas général :

Proposition 4.8.2 *Le problème* HORN SAT *est dans* P.

Démonstration. Si φ est une formule de Horn. On notera V l'ensemble de ses variables et C l'ensemble de ses clauses. On représente les clauses de Horn alternativement sous forme ensemblistes ou implicatives par $c = \{\neg p_1, ..., \neg p_k, q\}$ ou par :

$$c \equiv p_1 \wedge \cdots \wedge p_k \to q,$$

On note $c^+ = \{q\}$ et $c^- = \{p_1, ..., p_k\}$. On dit qu'une clause est unitaire positive si $c = c^+$. On note C_U l'ensemble des clauses unitaires positives de φ. On pose enfin, $C' := C \backslash C_U$. L'algorithme 1 teste la satisfaisabilité d'une formule de Horn en temps polynomial.

Algorithm 1 Horn

1: **Entrées :** C', C_U
2: $d \leftarrow 1$
3: **while** $C_u \neq \emptyset \wedge d = 1$ **do**
4: Soit $x \in C_U$
5: **for** $c \in C'$ **do**
6: **if** $x \subseteq c^-$ **then**
7: $c \leftarrow c \backslash x$
8: **if** $c = c^+$ **then** $C_U \leftarrow C_U \cup \{c\}$; $C' \leftarrow C' \backslash \{c\}$
9: **end if**
10: **if** $c = \emptyset$ **then** $d := 0$
11: **end if**
12: **end if**
13: $C_U \leftarrow C_U \backslash \{x\}$
14: **end for**
15: **end while**
16: **if** $d = 1$ **then** accepter
17: **else**refuser
18: **end if**

L'algorithme est simple et peut s'interpréter de la façon suivante. Toute variable apparaissant dans une clause positve unitaire est affectée à "vrai". Puis, on élimine x des clauses c où elle apparait en prémisse, i.e. dans c^-. A chaque itération, soit une clause finit par se restreindre à sa partie positive et dans

ce cas, la clause rejoint les clauses unitaires et on continue. Soit il existe une clause entièrement négative dont toutes les variables ont finalement été éliminées (donc instanciées à vrai indirectement). Dans ce cas, la formule n'est pas satisfiasable et on conclut.A chaque itération dans la boucle principale (qui prend un temps linéaire : on teste chaque clause de C'), au moins une variable disparait. L'algorithme fonctionne donc en temps polynomial (quadratique, en fait).

■

On peut même aller plus loin :

Proposition 4.8.3 *Le problème* HORN SAT *est* P*-complet.*

Démonstration. On obtient le résultat de dureté par réduction à partir du problème de décision d'un langage par une machine de Turing déterministe polynomiale. On fait exactement la même preuve que pour le Théorème 4.6.1 au sujet de laquelle on peut dresser les remarques suivantes.

La formule construite $\Phi_{M,w}$ est composée de clauses des types $I_1, \ldots, I_3$, $C_1, \ldots, C_7$, T'_1, T'_2 et FINAL. Parmi elles, seules C_1,C_3,C_5 et T'_2 ne sont pas de Horn.

Il est facile de voir que certaines clauses sont redondantes. Les clauses de type C_1, C_3, C_5 sont des conséquences (se montre par induction sur t pour l'ensemble et pour tout i) de C_7, I_1, I_2, I_3, T'_1 et T'_2. Pour C_1 : si à l'instant 0, la case i contient un caractère, qu'elle n'est pas modifiée lorsque la tête de lecture n'est pas sur elle, le seul moment où elle pourrait être vide est après que la tête de lecture soit passée sur elle, mettons en t. Or dans ce cas, une transition (dépendant de l'état) est tirée à l'instant t et le contenu de la case i est modifié (même si c'est pour contenir le caractère $\sqcup$). Les cas de C_3 et C_5 sont analogues et plus faciles.

Enfin, puisque notre machine M est déterministe, son nombre de choix possible de transitions est $c = 1$, ce qui transforme T'_2 en $Choix(t, 1)$ (on peut même s'en passer). Toutes les clauses sont donc de Horn.

On rappelle qu'une 2-clause est une clause comportant au plus deux littéraux. On peut définir une autre variante de SAT.

2-SAT
Entrée : Une formule propositionnelle φ constituée de 2-clauses
Question : φ est-elle satisfaisable ?

La proposition suivante est vraie.

Proposition 4.8.4 2SAT *est dans* NL *(donc dans* P*).*

Démonstration. On donne une preuve de l'appartenance à P. On peut appliquer la méthode de résolution pour toutes paires de clauses ayant une variable en commun. Cette méthode appliquée à des clauses de longueurs 2 produit des clauses résolvantes de longueur 2 au plus aussi et donc qu'au bout d'un nombre quadratique "d'étapes" au pire, soit la clause $\perp$ aura été produite soit plus aucune nouvelle clause n'est produite par résolution.

Une autre variante de SAT est la suivante. Une clause est *affine* si elle est de la forme :

$$p_1 \oplus p_2 \oplus \cdots \oplus p_n = 1 \text{ ou } p_1 \oplus p_2 \oplus \cdots \oplus p_n = 0$$

où $\oplus$ est le "ou disjonctif" i.e. l'addition modulo 2 : $0 \oplus 0 = 1 \oplus 1 = 0$, $0 \oplus 1 = 1 \oplus 0 = 1$. Une formule est affine si elle est constituée de clauses affines.

AFFINE SAT
Entrée : Une formule propositionnelle φ affine
Question : φ est-elle satisfaisable ?

Proposition 4.8.5 AFFINE SAT *est dans* P.

Démonstration. Au vu des lois qui régissent le $\oplus$, trouver un modèle à une formule affine revient à résoudre un système d'équation linéaire dans le corps à deux éléments. Ceci peut être fait en temps polynomial (par la méthode de Gauss, par exemple).

4.8.1 Un théorème dichotomique

Une formule φ est dite 0-valide (resp. 1-valide) si l'assignation qui affecte 0 (resp. 1) à toute variable est un modèle de φ. On a déjà vu Horn et affine. Lorsque les clauses sont des 2-clauses, on dit qu'elles sont *bijonctives*. Enfin, le cas symétrique de Horn (au plus un littéral négatif) est appelé anti-Horn.

On appelle $\mathrm{SAT}(\mathscr{F})$ le problème SAT restreint aux formules φ dont *toutes* les clauses sont de type $\mathscr{F}$ (par exemple affine, Horn, etc) [9].

Le résultat suivant (assez long à démontrer) montre que la complexité des problèmes de satisfaisabilité obéit à une certaine forme de "tout ou rien".

Théorème 4.8.1 (Schaefer'78) *Lorsque $\mathscr{F}$ est affine, 0-valide, 1-valide, Horn, anti-Horn ou bijonctif le problème $\mathrm{SAT}(\mathscr{F})$ est dans P. Dans tous les autres cas, le problème $\mathrm{SAT}(\mathscr{F})$ est NP-complet.*

Exercice 19 Montrer que le problème de satisfaisabilité d'une formule propositonnelle dont les clauses sont soit affines soit de Horn est NP-complet. En d'autre termes, montrer que pour toute formule CNF, φ, on peut trouver une formule CNF, ψ dont toutes les clauses sont soit de Horn soit affines et telle que φ est satisfaisable ssi ψ est satisfaisable.

Le résultat précédent (que l'on ne démontre pas non plus) est loin d'être acquis comme l'indique le théorème suivant.

Théorème 4.8.2 (Ladner'75) *Si $\mathrm{P} \neq \mathrm{NP}$, il y a des problèmes de NP qui ne sont ni dans P, ni NP-complets.*

9. Dans l'appendice de ce chapitre, on rentre un peu plus en détails sur ce genre de propriétés

Chapitre 5

Retour sur la complexité en espace

On revient sur la complexité en espace pour donner un exemple de problème complet pour NL et montrer que dans ce cadre le non déterminisme n'apporte pas de puissance de calcul essentielle.

5.1 Un problème NL-complet

5.1.1 L'accessibilité dans un graphe

Dans cette section on va montrer, petit à petit, un résultat surprenant sur les classes en espace non déterministe : leur clôture par complémentation. On le fait pour la plus petite d'entre elles NL par le biais d'un problème naturel appelé ACCESSIBILITÉ dont on montre :

- qu'il appartient à NL et co−NL,
- qu'il est NL-complet.

La définition de ce problème est la suivante.

ACCESSIBILITÉ
Entrée : Un graphe $G = \langle V, E\rangle$ orienté, deux sommets s et t de V.
Question : existe-t'il un chemin (orienté) reliant s à t i.e. une suite d'arêtes $e_1 = (s, y_1), e_2 = (x_2, y_2), \ldots, e_k = (x_k, t)$ telles que : $y_1 = x_2$, $y_2 = x_3$, ..., $y_{k-1} = x_k$?

Le premier résultat est assez facile à montrer.

Proposition 5.1.1 *Le problème* ACCESSIBILITÉ *appartient à* NL.

Démonstration. Soit $G = \langle V, E\rangle$ un graphe orienté, s et t dans V et $|V| = n$. On va décrire un algorithme non déterministe qui parcourt le graphe à partir de s de voisin en voisin jusqu'à rencontrer (ou pas) le sommet t. La particularité de cet algorithme est qu'il ne se souvient pas (et ne le peut pas pour des raisons de mémoire) des endroits déjà visités.

Algorithm 2 Access

1: $i \leftarrow 0$, $tmp := s$
2: **while** $i \leq n$ **do**
3: Choisir de façon non déterministe un voisin de tmp i.e. x t.q. $E(tmp, x)$
4: **if** $x = t$ **then** *accepter*
5: **else** $tmp \leftarrow x$
6: **end if**
7: $i \leftarrow i + 1$
8: **end while**
9: *rejeter*

L'algorithme Access(G, s, t) accepte si, à un moment du calcul, il existe x tel que $x = t$. Comme on a procédé par voisin successif, il existe alors un chemin menant de s à t. L'algorithme refuse si au bout de

n étapes on a toujours pas rencontré t quelques soient les choix effectués. Or, s'il existait un chemin de s à t, il en existerait un de longueur inférieure ou égale à n. L'algorithme refuse donc bien quand il n'y a pas de chemin de s à t.

Dans l'algorithme, on peut supposer que les sommets de V sont numérotés de 1 à $|V| = n$. A chaque étape, l'algorithme $Access$ doit se souvenir de i, de n, du sommet courant tmp et passer à un sommet suivant. Chacun des objets peut se coder par un entier de 1 à n et nécessite donc, au plus, $O(\log n)$ bits. D'où la borne en espace.

Proposition 5.1.2 *Le problème* ACCESSIBILITÉ *est* NL*-complet.*

Démonstration (idée). Soit M une machine de Turing non déterministe telle que $L(M) \in$ NL. Soit w une entrée de M telle que $|w| = n$. M fonctionne en espace $O(\log n)$ sur l'entrée w. On fait en sorte que M n'ait toujours qu'une seule configuration intiale C_I et finale C_F (pour cette dernière, il suffit simplement de forcer la machine à effacer son espace de travail et ramener ses têtes de lecture en débuts de bandes avant la fin de tout calcul acceptant).

On sait qu'une telle machine a un nombre de configurations différentes borné par un certain polynome $P(n)$. On construit un graphe $G = \langle V, E\rangle$ dont les sommets sont les configurations C possibles de calculs de M sur w et dont les arêtes témoignent de la relation d'accessibilité en une étape i.e. :

$$\forall C_1, C_2 \in V,\ E(C_1, C_2) \text{ ssi } C_1 \vdash_M^1 C_2$$

On vérifie facilement que M accepte w ssi la configuration finale C_F est accessible à partir de la configuration initiale C_0 c'est à dire s'il existe un chemin dans notre graphe menant de C_I à C_F. Cela conclut la preuve

5.1.2 Le théorème de Savitch

On va montrer le théorème suivant sur les classes en espace.

Théorème 5.1.1 (Savitch) *Soit* $S : \mathbb{N} \to \mathbb{N}$ *constructible telle que* $S(n) \leq \log n$*,*

$$\text{NSPACE}(S(n)) \subseteq \text{DSPACE}(S^2(n)).$$

Démonstration. Soit M une machine non déterministe et $L(M) \in \text{NSPACE}(S(n))$. Il existe donc c tel que, sur toute entrée w de taille n, M travaille en espace borné par $cS(n)$. Le temps de calcul est borné par $2^{dS(n)}$ pour un certain d.

Soit C l'ensemble des configurations distinctes possibles de la machine M sur l'entrée w. Celles-ci sont en nombre au plus $2^{dS(n)}$. On décrit une procédure $ConfAccess(c_1, c_2, t)$ qui teste si deux configurations c_1 et c_2 sont accessibles en au plus t étapes de calcul. Le résultat sera donc obtenu en appelant $ConfAccess(c_I, c_F, 2^{dS(n)})$ où c_I (resp. c_F) est la configuration initiale (resp. finale). Il reste à montrer comment calculer ConfAccess$(c_I, c_F, 2^{dS(n)})$ par un algorithme déterministe fonctionnant en espace $O(S^2(n))$. On le décrit par une procédure récursive.

L'algorithme ConfAccess(c_1, c_2, t) s'appelle récursivement avec une valeur de t qui est divisée par 2 à chaque étape. Avant l'appel, il doit donc empiler les variables de son propre environnement soit c_1, c_2 et t (qui sont restaurées au retour de l'appel récursif). L'ensemble prend un espace $O(S(n))$. En partant de $t = 2^{dS(n)}$, il fait donc un nombre d'appel récursif sur chaque branche de l'arbre de l'ordre de $\log 2^{dS(n)} = O(S(n))$. Au pire moment, l'espace nécessaire est de l'ordre de $O(S(n).S(n)) = O(S^2(n))$. Noter, pour conclure qu'en supposant $S(n)$ constructible, on s'est simplifié la vie : on peut utiliser $S(n)$ et $2^{dS(n)}$ comme arguments dans les appels.

Noter que le Théorème 5.1.1 implique aussi que ACCESSIBILITÉ $\in \text{DSPACE}(\log^2(n))$. Comme corollaire du théorème de Savitch, il vient immédiatement :

Corollaire 5.1.2 PSPACE = NPSPACE.

Algorithm 3 ConfAccess

1: **Entrée :** c_1, c_2, t
2: **if** $t = 1$ **then**
3: **if** $c_1 = c_2$ **then** *Accepter*
4: **else if** $c_1 \vdash_M^1 c_2$ **then** *Accepter*
5: **else** *Rejeter*
6: **end if**
7: **end if**
8: **for all** $c_3 \in C$ **do**
9: **if** ConfAccess$(c_1, c_3, \lceil \frac{t}{2} \rceil)$ et ConfAccess$(c_3, c_2, \lceil \frac{t}{2} \rceil)$ **then** *Accepter*
10: **end if**
11: **end for**
12: *Rejeter*

5.1.3 Clôture par complémentaire de NL

Soit A un langage (sur un certain alphabet Σ), on appelle $\bar{A} = \{x : x \notin A\}$ le complémentaire de A. Pour toute classe de complexité $\mathscr{C}$, on appelle $co-\mathscr{C}$ la classe complémentaire i.e. pour tout langage A :

$$A \in \mathscr{C} \iff \bar{A} \in co-\mathscr{C}.$$

Proposition 5.1.3 *Si A est* NL*-complet (par réduction logarithmique) et $A \in$ co−NL alors* NL = co−NL

Démonstration. Supposons que quelque soit $B \in$ NL, $B \leq_{\mathrm{L}} A$. Dans ce cas, il existe $f \in$ LOGSPACE telle que :

$$\forall x \in \Sigma^*, B(x) \iff A(f(x)).$$

On a donc aussi :

$$\forall x \in \Sigma^*, \bar{B}(x) \iff \bar{A}(f(x)),$$

montrant que $\bar{A}$ est co−NL-complet. A étant dans co−NL, il existe donc $g \in$ LOGSPACE telle que :

$$\forall x \in \Sigma^*, A(x) \iff \bar{A}(g(x)).$$

En mettant bout à bout les réductions :

$$\forall x \in \Sigma^*, B(x) \iff \bar{A}(g(f(x))).$$

Cela nous donne un algorithme dans co−NL pour B dont on a supposé qu'il était dans NL. D'où le résultat.

De la proposition 5.1.2, on peut déduire ce qui suit par réduction.

Proposition 5.1.4 *Le problème* 2-SAT *est* co−NL*-complet.*

Démonstration. On commence par montrer que 2-SAT est dans co−NL. On le fait par réduction de 2-$\overline{\text{SAT}}$ à ACCESSIBILITÉ. Soit φ une formule propositionnelle de la forme $\bigwedge_{i=1}^{k} C_i$ où chaque $C_i \equiv x_1^i \vee x_2^i$ est une 2-clause. L'ensemble de varaible de φ est $\{p_1, \ldots, p_n\}$. On construit un graphe $G = (V, E)$ dont l'ensemble des sommets V est $\{p_1, \ldots, p_n, \neg p_1, \ldots, \neg p_n\}$. Pour chaque clause $C \equiv x_1 \vee x_2$ pouvant être vue comme $(\neg x_1 \to x_2)$ et $(\neg x_2 \to x_1)$ on crée deux arêtes (orientées) $E(\neg x_1, x_2)$ et $E(\neg x_2, x_1)$.

Soient x et y deux sommets, on notera $x \twoheadrightarrow y$ l'existence d'un chemin de x vers y dans G. On montre que :

φ est non satisfaisable ssi il existe un sommet x tel que $x \twoheadrightarrow \neg x$ et $\neg x \twoheadrightarrow x$.

S'il existe x tel que $x \twoheadrightarrow \neg x$ et $\neg x \twoheadrightarrow x$ alors dans tout assignement des variables de φ comme x et $\neg x$ prennent des valeurs inverses, une des deux chaines d'implications est de la forme $1 \twoheadrightarrow 0$. Il y a

donc deux sommets sur le chemin tels que $p \rightarrow q$ et $p = 1$ et $q = 0$. La clause $C \equiv \neg p \vee q$ ne peut donc être vraie et l'assignement n'est pas un modèle de φ.

Réciproquement, supposons qu'il n'existe pas x tel que $x \rightarrow\!> \neg x$ et $\neg x \rightarrow\!> x$[1]. On peut construire un assignement I pas à pas de la façon suivante. Soit x un sommet du graphe non encore affecté tel qu'il n'y a pas de chemin de x vers $\neg x$ (sinon on prend $\neg x$ car on ne peut avoir $x \rightarrow\!> \neg x$ et $\neg x \rightarrow\!> x$) et $S(x)$ l'ensemble des points atteignables à partir de x. On affecte x et tout point de $S(x)$ à 1 dans I et les négations de ces points à 0. Peut-il y avoir dans cet assignement un chemin de x vers à la fois y et $\neg y$ (qui rendrait l'assignement invalide à ce stade) ? si tel était le cas, on aurait $x \rightarrow\!> y$, $x \rightarrow\!> \neg y$ et donc aussi $y \rightarrow\!> \neg x$ ce qui implique $x \rightarrow\!> \neg x$ (et qui contredit notre choix de x). De même, une variable x assignée à 1 à cette étape ne peut atteindre une variable déjà assignée à 0 (sinon, en tant que prédécesseur de cette variable (ou successeur de sa négation), elle devrait déjà avoir la valeur 0). Aucune clause implicative n'est donc invalide, l'assignement est bien satisfaisant.

La complétude de 2-$\bar{\text{S}}$AT s'obtient par réduction à partir de ACCESSIBILITÉ là aussi. Soit $G = (V, E)$ et s et t, on construit simplement les clauses s, $\neg t$ et $x \rightarrow y$ pour toute arête $E(x, y)$.

Logiquement, on appelera donc ACCE$\bar{\text{S}}$SIBILITÉ le complémentaire du problème ACCESSIBILITÉ i.e. on accepte les instances $G = \langle V, E \rangle$, s et t telles qu'il n'y a pas de chemins de s à t dans G. De façon surprenante, on peut montrer le résultat suivant.

Théorème 5.1.3 ACCE$\bar{\text{S}}$SIBILITÉ *est dans* NL.

Démonstration. On construit un algorithme en deux parties. Commençons par la plus facile. Dans l'algorithme Vérif(G, s, t, N_s) détaillé après, on suppose que l'on connait à la fois le graphe G, les deux sommets concernés s et t mais aussi N_n le nombre de sommets accessibles à partir de s. Dans ce cas, on peut tester en temps non déterministe logarithmique si t est accessible à partir de s.

Algorithm 4 Vérif

```
Entrée : G, s, t, N_s
d ← 0
for all v ∈ V\{t} do
   Choisir de façon non déterministe un chemin ch de longueur inférieur ou égal à n à partir de s. Si
   v n'apparait à aucun moment dans ch alors passer au sommet v suivant.
     d ← d+1
end for
if d = N_s then Accepter
else rejeter
end if
```

L'algorithme Vérif(G, s, t, N_s) passe en revue l'ensemble des points de V (sauf t) et incrémente un compteur d à chaque fois que l'un d'entre eux est accessible à partir de s[2]. A la fin, on compare N_s à d et on accepte s'ils sont égaux ce qui signifient que tous les sommets accessibles sont repérés, sans tester t, et donc que t ne peut en faire partie. Cet algorithme ne doit se souvenir à chaque étape que de v, d et des sommets intermédiaires (dans les chemins). Il fonctionne clairement en espace logarithmique.

Il nous reste maintenant à calculer le nombre N_s en construisant inductivement une suite de nombre N_s^i ($i \leq n$) du nombre de sommets accessibles par des chemins de longueurs i à partir de s. On aura $N_s^n = N_s$. Ceci est fait par la procédure NbSommetAccess(G, s, t)

Examinons d'abord la boucle la plus interne et l'instruction conditionnelle qui suit (étape 7 à 14). On dit qu'un point u est i-accessible s'il existe un chemin de longueur inférieure ou égale à i de s à u. Pour chaque v fixé, on sort dans un état rejetant des instructions internes si, après une itération complète on a $d \neq N_s^i$ i.e. que l'on a pas visité tous les sommets u i-accessibles. En d'autres termes, on continue dans cette partie si, soit v est tel que $E(u, v)$ avec u i-accessible (et dans ce cas, on incrémente N_s^{i+1}), soit on a visité tous les u i-accessibles possibles (auquel cas, on passe au sommet v suivant). Le phénomène doit

1. même si un des deux chemins peut exister pour tout x
2. pour tester l'existence d'un chemin, on choisit en fait des sommets v_i t.q. $s = v_0$ et $E(v_i, v_{i+1})$ et on teste à chaque étape si v en fait partie. On refuse si au bout de n étapes (au plus) on ne l'a pas trouvé.

Algorithm 5 NbSommetAccess

1: **Entrée :** G, s, t
2: $N_s^0 \leftarrow 1$
3: **for all** $i \in \{0, \dots, n-1\}$ **do**
4: $N_s^{i+1} \leftarrow 1$
5: **for all** $v \in V \setminus \{s\}$ **do**
6: $d \leftarrow 0$
7: **for all** $u \in V$ **do**
8: Choisir de façon non déterministe un chemin ch (de longueur $\leq i$) à partir de s. Si u n'apparait à aucun moment dans ch alors passer au sommet u suivant.
9: $d \leftarrow d+1$
10: **if** $E(u, v)$ **then** $N_s^{i+1} \leftarrow N_s^{i+1} + 1$ et passer au sommet suivant v (étape 6)
11: **end if**
12: **end for**
13: **if** $d \neq N_s^i$ **then** *rejeter*
14: **end if**
15: **end for**
16: **end for**
17: Accepter et sortir N_s^n

donc se passer pour tout v (excepté s). Tout sommet $(i+1)$-accessible étant successeur d'un sommet i-accessible, on sort de la boucle sans refuser quand on a passé en revue trouvé l'ensemble des sommets $i+1$-accessibles. Ce nombre est stocké dans N_s^{i+1} qui est incrémenté à chaque nouveau sommet trouvé.

A tout moment, cet algorithme n'a besoin de se souvenir que de i, u, v et des sommets intermédiaires dans le calcul de chemins. Il réalise aussi des opérations d'incrémentation. Tout ceci nécessite un espace logarithmique, au plus. Remarquer aussi, que l'on a pas besoin d'une N_s^i pour tout i. Avec seulement deux variables réutilisables on peut faire le même travail (même si c'est moins lisible).

Il reste, pour conclure la preuve à combiner les deux algorithmes, d'abord NbSommetAccess(G, s, t) qui renvoit N_s puis Vérif(G, s, t, N_s).

Les résultats précédents concernant le problème ACCESSIBILITÉ ou son complémentaire impliquent que NL est close par complémentaire : on a un problème qui est à la fois dans co−NL et NL-complet (cf. proposition 5.1.3).

Théorème 5.1.4 (Immerman'88,Szelepscényi'89) NL = co−NL.

Le résultat se généralise à toute classe de complexité.

Théorème 5.1.5 *Soit $S : \mathbb{N} \to \mathbb{N}$ constructible. On a :*

$$\text{NSPACE}(S(n)) = co{-}\text{NSPACE}(S(n))$$

Démonstration. Ce genre de résultat se prouve par une technique appelée "rembourrage". Soit $A \in$ NSPACE$(S(n))$ et M fonctionnant en espace $S(n)$ reconnaissant A i.e. $L(M) = A$. Soit A^{remb} le langage suivant :

$$A^{remb} = \{x10^{2^{S(n)}} : \; x \in A \text{ et } |x| = n\}.$$

On va montrer que $A \in co{-}$NSPACE$(S(n))$ en utilisant A^{remb} et l'égalité NL = co−NL. Soit l'algorithme N non déterministe suivant qui, sur toute entrée x :

– Construit $w = x10^{2^{S(n)}}$ où $|x| = n$.
  Cette étape se fait en espace déterministe $O(\log(n + 2^{S(n)})) = O(S(n))$ (si $S(n) > \log n$, ce qui nous intéresse ici). Pour cela, il suffit de maintenir un compteur d de 0 à $2^{S(n)}$ sur le ruban de travail et de rajouter un 0 à la sortie w à chaque étape ($S(n)$ est constructible).

– on accepte w ssi M accepte x.
  Cette deuxième étape nécessite un espace non déterministe $O(S(|x|)) = O(\log|w|)$ par exemple en simulant M directement. En d'autre terme, si on se donne un mot w de la forme $x10^{2^{S(n)}}$ reconnaitre que x est dans A peut se faire dans NL et donc dans co−NL par le Théorème 5.1.4. Il existe donc un algorithme non déterministe universel reconnaissant l'appartenance d'un mot x à A en espace $O(\log|w|) = O(S(|x|))$.

L'algorithme ci-dessus, dont les deux étapes sont bornés en espace par $O(S(n))$ permet de montrer que $A \in co-\text{NSPACE}(S(n))$.

Table des matières

1 Modèles de calcul **3**
1.1 Fonctions récursives primitives . . . 3
1.1.1 Exemples de fonctions récursives primitives . . . 4
1.1.2 Propriétés de clôtures . . . 5
1.1.3 Prédicats définissables au premier ordre par quantification bornée . . . 7
1.1.4 Premiers codages . . . 7
1.2 Les fonctions récursives . . . 11
1.2.1 Évaluation des fonctions récursives primitives . . . 11
1.2.2 Fonction d'Ackermann . . . 12
1.2.3 Fonctions partielles récursives . . . 14
1.3 Fonctions calculables par machines à registres. . . . 17
1.3.1 Programmes goto. . . . 17
1.3.2 De nouvelles instructions. . . . 18
1.3.3 Programmes structurés. . . . 19
1.4 Les fonctions récursives sont calculables par machines. . . . 22
1.4.1 Les fonctions récursives partielles. . . . 23
1.4.2 Les fonctions récursives primitives. . . . 24
1.5 Les fonctions calculables par machine sont récursives. . . . 24
1.5.1 Machine, état d'une machine. . . . 24
1.5.2 Le calcul. . . . 25
1.5.3 Thèse de Church . . . 29
1.5.4 Forme normale de Kleene. . . . 30
1.5.5 Fonctions universelles. . . . 31
1.6 Problèmes indécidables . . . 34
1.6.1 Ensembles récursivement énumérables . . . 34
1.6.2 Prédicats et problèmes . . . 36
1.6.3 Problème de l'arrêt : la méthode diagonale . . . 36
1.6.4 Ensembles séparables, théorème de Rice . . . 37
1.7 Réductions . . . 39
1.7.1 Réduction "many-one" . . . 39
1.7.2 Réduction de Turing . . . 40
1.8 Une méthode des priorités . . . 41

2 Modèles de calcul : compléments **45**
2.1 Machines à accès indirect . . . 45
2.2 Machines de Turing . . . 47
2.2.1 Machines de Turing déterministes . . . 47
2.2.2 Exemples . . . 48
2.2.3 Extensions du modèle . . . 49
2.3 Les RAM et les machines de Turing calculent les mêmes fonctions . . . 49

3 Problèmes de décision en logique **51**
3.1 Rappels : signature, structure, formule . . . 51
3.2 Problèmes de décision logique indécidables . . . 51
3.3 Méthodes de décision : les jeux d'Ehrenfeucht-Fraïssé . . . 54
3.3.1 Jeux et va-et-vient . . . 54
3.3.2 Formules et types . . . 55
3.3.3 Théorie du premier ordre monadique . . . 57
3.3.4 Le cas d'une seule fonction unaire . . . 58
3.3.5 Théorie de l'ordre total discret . . . 59
3.4 L'arithmétique de Presburger . . . 60

4 Rudiments de Complexité **63**
4.1 Modèles de calcul . . . 63
4.1.1 Machines de Turing non déterministes . . . 63
4.1.2 Arbres de calcul, langages reconnaissables . . . 63
4.1.3 Fonctions . . . 64
4.2 Mesures de complexité en temps et en espace . . . 65
4.2.1 Classes de complexité en temps et en espace . . . 65
4.3 Inclusions et hiérarchies entre les classes . . . 66
4.3.1 Fonctions constructible en temps et espace . . . 66
4.3.2 Relations d'inclusion entre les classes . . . 66
4.3.3 Hiérarchies en temps et espace . . . 67
4.4 Définition alternative de NP . . . 68
4.5 Réductions entre problèmes . . . 69
4.6 NP-complétude de SAT . . . 70
4.7 Quelques exemples de problèmes NP-complets . . . 73
4.8 Des variantes plus faciles de la satisfaisabilité . . . 73
4.8.1 Un théorème dichotomique . . . 76

5 Retour sur la complexité en espace **77**
5.1 Un problème NL-complet . . . 77
5.1.1 L'accessibilité dans un graphe . . . 77
5.1.2 Le théorème de Savitch . . . 78
5.1.3 Clôture par complémentaire de NL . . . 79

6 Arithmétique **83**
6.1 Définissabilité dans $\mathbb{N}$ . . . 83
6.1.1 formules et ensembles Σ_0 et Σ . . . 83
6.1.2 Fonctions récursives et Σ-définissabilité . . . 85
6.1.3 Hiérarchie arithmétique . . . 88
6.1.4 Ensembles arithmétiques . . . 91
6.2 Le codage des formules . . . 92
6.2.1 Codage des arbres binaires . . . 93
6.2.2 Codage des expressions arithmétiques . . . 94
6.3 La vérité dans l'arithmétique n'est pas définissable . . . 95