# Calculabilité

INF110 (Logique et Fondements de l'Informatique)

David A. Madore
Télécom Paris
david.madore@enst.fr

2023–2024

http://perso.enst.fr/madore/inf110/transp-inf110.pdf

Git: a90f2fd Mon Jan 22 13:08:29 2024 +0100

## Plan

**Plan** [transp. 2]

## Table des matières



## 1 Introduction

**Qu'est-ce que la calculabilité ?** [transp. 3]

▶ À l'interface entre **logique mathématique** et **informatique théorique**
  ▶ née de préoccupations venues de la logique (Hilbert, Gödel),
  ▶ à l'origine des 1$^{\text{ers}}$ concepts informatiques ($\lambda$-calcul, machine de Turing).

▶ But : étudier les limites de ce que **peut ou ne peut pas faire un algorithme**
  ▶ sans limite de ressources (temps, mémoire juste « finis »),

- ▶ sans préoccupation d'efficacité ($\neq$ complexité, algorithmique),
- ▶ y compris résultats négatifs (« *aucun* algorithme ne peut... »),
- ▶ voire relatifs (calculabilité relative),
- ▶ admettant diverses généralisations (calculabilité supérieure).

**Quelques noms** [transp. 4]

- ▶ Muḥammad ibn Mūsá al-Ḫwārizmī (v.780–v.850) : $\rightsquigarrow$« algorithme »
- ▶ Blaise Pascal (1623–1662) : machine à calculer $\rightsquigarrow$automates
- ▶ Charles Babbage (1791–1871) : *Analytical Engine* (Turing-complète !)
- ▶ Ada (née Byron) Countess of Lovelace (1815–1852) : programmation
- ▶ Richard Dedekind (1831–1916) : définitions primitives récursives
- ▶ David Hilbert (1862–1943) : *Entscheidungsproblem* (décider la vérité)
- ▶ Jacques Herbrand (1908–1931) : fonctions générales récursives
- ▶ Kurt Gödel (1906–1978) : incomplétude en logique
- ▶ Haskell Curry (1900–1982) : logique combinatoire, lien preuves-typage
- ▶ Alonzo Church (1903–1995) : $\lambda$-calcul
- ▶ Alan M. Turing (1912–1954) : machine de Turing, problème de l'arrêt
- ▶ Emil Post (1897–1954) : ensembles calculablement énumérables
- ▶ Stephen C. Kleene (1909–1994) : $\mu$-récursion, th. de récursion, forme normale

**Fonction calculable** [transp. 5]

« Définition » : une fonction $f$ est **calculable** quand il existe un algorithme qui

- ▶ prenant en entrée un $x$ du domaine de définition de $f$,
- ▶ **termine en temps fini**,
- ▶ et renvoie la valeur $f(x)$.

Difficultés :

- ▶ Comment définir ce qu'est un algorithme ?
- ▶ Quel type de valeurs (acceptées et renvoyées) ?
- ▶ Et si l'algorithme ne termine pas ?
- ▶ Distinction entre intention (l'algorithme) et extension (la fonction).

**Sans préoccupation d'efficacité** [transp. 6]

▶ La calculabilité *ne s'intéresse pas à l'efficacité* des algorithmes qu'elle étudie, uniquement leur **terminaison en temps fini**.

P.ex. : pour savoir si $n$ est premier, on peut tester si $i \times j = n$ pour tout $i$ et $j$ allant de 2 à $n-1$. (Hyper inefficace ? On s'en fiche.)

▶ La calculabilité *n'a pas peur des grands entiers.*

P.ex. : **fonction d'Ackermann** définie par :

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

définition algorithmique (par appels récursifs qui terminent), donc calculable.

Mais $A(2,6,3) = 2^{2^{2^{2^2}}} = 2^{2^{65\,536}}$ et $A(2,4,4) = A(2,65\,536,3)$ est inimaginablement grand (et que dire de $A(100,100,100)$ ?).
⇒ Ingérable sur un vrai ordinateur.

**Approches de la calculabilité [transp. 7]**

▶ Approche informelle : **algorithme = calcul finitiste** mené par un humain ou une machine, selon des instructions précises, en temps fini, sur des données finies

▶ Approche pragmatique : tout ce qui peut être fait sur un langage de programmation « Turing-complet » (Python, Java, C, Caml...) idéalisé

- ▶ sans limites d'implémentation (p.ex., entiers arbitraires !),
- ▶ sans source de hasard ou de non-déterminisme.

▶ Approches formelles, p.ex. :

- ▶ fonctions générales récursives (Herbrand-Gödel-Kleene),
- ▶ $\lambda$-calcul (Church) ($\leftrightarrow$ langages fonctionnels),
- ▶ machine de Turing (Turing),
- ▶ machines à registres (Post...).

▶ **« Thèse » de Church-Turing** : *tout ceci donne la même chose.*

**Thèse de Church-Turing [transp. 8]**

▶ **Théorème** (Post, Turing) : les fonctions (disons $\mathbb{N} \dashrightarrow \mathbb{N}$) **(1)** générales récursives, **(2)** représentables en $\lambda$-calcul, et **(3)** calculables par machine de Turing, coïncident toutes.
⇒ On parle de *calculabilité au sens de Church-Turing.*

▶ **Observation** : tous les langages de programmation informatiques généraux usuels, idéalisés, calculent aussi exactement ces fonctions ($\rightarrow$ « Turing-complets »).

▶ **Thèse philosophique** : la calculabilité de C-T définit précisément la notion d'algorithme finitiste.

▶ **Conjecture physique** : la calculabilité de C-T correspond aux calculs réalisables mécaniquement dans l'Univers (en temps/énergie finis mais illimités).
↑ (même avec un ordinateur quantique)

Pour toutes ces raisons, le sujet mérite d'être étudié !

### Trois grandes approches [transp. 9]

On va décrire trois approches des (mêmes !) fonctions calculables au sens de Church-Turing, et esquisser leur équivalence :

- Les **fonctions générales récursives** sont mathématiq$^{\text{t}}$ plus commodes :
  - « tout est un entier » (fonctions $\mathbb{N}^k \dashrightarrow \mathbb{N}$),
  - définition inductive, numérotation associée.
- Les **machines de Turing** représentent des ordinateurs très simples :
  - travaillent sur une « bande » illimitée a priori (mémoire),
  - aspect algorithmique évident, plus proche d'un « vrai » ordinateur,
  - approche la plus commode pour la complexité (pas considérée ici).
- Le **$\lambda$-calcul** pur non typé est un système symbolique :
  - proche des langages de program$^{\text{tion}}$ fonctionnels (Lisp, Haskell, OCaml...),
  - plus facile à « programmer » réellement, mais nombreuses subtilités.

### Données finies [transp. 10]

Un algorithme travaille sur des **données finies**.

Qu'est-ce qu'une « donnée finie » ? Tout objet représentable informatiquement : booléen, entier, chaîne de caractères, structure, liste/tableau de ces choses, ou même plus complexe (p.ex., graphe).

→ Comment y voir plus clair ?

Deux approches opposées :

- **typage** : distinguer toutes ces sortes de données,
- **codage de Gödel** : tout représenter comme des entiers !

Le typage est plus élégant, plus satisfaisant, plus proche de l'informatique réelle, on en reparlera.
Le codage de Gödel simplifie l'approche/définition de la calculabilité (on étudie juste des fonctions $\mathbb{N} \dashrightarrow \mathbb{N}$).

### Codage de Gödel (« tout est un entier ») [transp. 11]

- Représenter **n'importe quelle donnée finie par un entier**.
- Codage des couples : par exemple,
$$\langle m, n\rangle := m + \frac{1}{2}(m+n)(m+n+1)$$
définit une bijection calculable $\mathbb{N}^2 \to \mathbb{N}$ (calculable dans les deux sens).
- Codage des listes finies : par exemple,
$$\langle\langle a_0, \dots, a_{k-1}\rangle\rangle := \langle a_0, \langle a_1, \langle \cdots, \langle a_{k-1}, 0\rangle + 1 \cdots\rangle + 1\rangle + 1$$
définit une bijection calculable $\{\text{suites finies dans } \mathbb{N}\} \to \mathbb{N}$ (avec $\langle\langle\rangle\rangle := 0$).
- Il sera aussi utile de représenter même les *programmes* par des entiers.
- Les détails précis du codage sont **sans importance**.
- Ne pas utiliser dans la vraie vie (hors calculabilité) !

**Fonctions partielles [transp. 12]**

▶ Même si on s'intéresse à des algorithmes qui **terminent**, la définition de la calculabilité *doit forcément* passer aussi par ceux qui ne terminent pas.
(Aucun langage Turing-complet ne peut exprimer uniquement des algorithmes qui terminent toujours, à cause de l'indécidabilité du problème de l'arrêt.)

▶ Lorsque l'algorithme censé calculer $f(n)$ ne termine pas, on dira que $f$ n'est pas définie en $n$, et on notera $f(n)\uparrow$. Au contraire, s'il termine, on note $f(n)\downarrow$.

▶ Notation : $f\colon \mathbb{N} \dashrightarrow \mathbb{N}$ : une fonction $D \to \mathbb{N}$ définie sur une partie $D \subseteq \mathbb{N}$.
▶ Notation : $f(n)\downarrow$ signifie « $n \in D$ », et $f(n)\uparrow$ signifie « $n \notin D$ ».
▶ Notation : $f(n)\downarrow = g(m)$ signifie « $f(n)\downarrow$ et $g(m)\downarrow$ et $f(n) = g(m)$ ».
▶ Convention : $f(n) = g(m)$ signifie « $f(n)\downarrow$ ssi $g(m)\downarrow$, et $f(n) = g(m)$ si $f(n)\downarrow$ ». (Certains préfèrent écrire $f(n) \simeq g(m)$ pour ça.)

▶ Convention : si $g_i(\underline{x})\uparrow$ pour un $i$, on convient que $h(g_1(\underline{x}), \ldots, g_k(\underline{x}))\uparrow$.

▶ Terminologie : une fonction $f\colon \mathbb{N} \to \mathbb{N}$ définie sur $\mathbb{N}$ est dite **totale**.
Une fonction totale est un *cas particulier* de fonction partielle !

**Terminologie à venir (avant-goût) [transp. 13]**

▶ Une fonction partielle $f\colon \mathbb{N} \dashrightarrow \mathbb{N}$ est dite **calculable** (partielle) lorsqu'il existe un algorithme qui prend $n$ en entrée et :

- ▶ termine (en temps fini) et renvoie $f(n)$ lorsque $f(n)\downarrow$,
- ▶ ne termine pas lorsque $f(n)\uparrow$.

▶ Une partie $A \subseteq \mathbb{N}$ est dite **décidable** lorsque sa fonction indicatrice $\mathbb{N} \to \mathbb{N}$

$$\mathbf{1}_A\colon n \mapsto \begin{cases} 1 & \text{si } n \in A \\ 0 & \text{si } n \notin A \end{cases}$$

est calculable (répondre « oui » ou « non » selon que $n \in A$ ou $n \notin A$).

▶ Une partie $A \subseteq \mathbb{N}$ est dite **semi-décidable** lorsque sa fonction partielle « semi-indicatrice » $\mathbb{N} \dashrightarrow \mathbb{N}$ (d'ensemble de définition $A$)

$$n \mapsto \begin{cases} 1 & \text{si } n \in A \\ \uparrow & \text{si } n \notin A \end{cases}$$

est calculable (répondre « oui » ou « ... » selon que $n \in A$ ou $n \notin A$).

**Point terminologique : « récursif » [transp. 14]**

Le mot « récursif » et ses cognats (« récursion », « récursivité ») a plusieurs sens *apparentés mais non identiques* :

- ▶ « récursif » = « défini par récurrence » (Dedekind 1888) → fonctions primitives récursives, générales récursives (cf. après) ;
- ▶ « récursif » = « calculable » (par glissement à cause de la définition de la calculabilité par les fonctions générales récursives) ;
- ▶ « récursif » = « faisant appel à lui-même dans sa définition » (appels récursifs, récursivité en informatique).

On va définir les fonctions « **primitives récursives** » (1er sens) et « **(générales) récursives** » (1er et aussi 2e sens) ci-après.

Pour le 3e sens, on dira « appels récursifs ».

# 2 Fonctions primitives récursives

**Fonctions primitives récursives : aperçu [transp. 15]**

▶ Avant de définir les fonctions générales récursives (≅ calculables), on va commencer par les **primitives récursives**, plus restreintes.
« primitive*ment* récursives » ?

▶ Historiquement antérieures à la calculabilité de Church-Turing.

▶ Pédagogiquement utile comme « échauffement ».

▶ À cheval entre calculabilité (**PR** est une petite classe de calculabilité) et complexité (c'est une grosse classe de complexité).

▶ Correspond à des programmes à **boucles bornées a priori**.

▶ Énormément d'algorithmes usuels sont p.r.

▶ Mais pas tous : p.ex. la fonction d'Ackermann n'est pas p.r.

**Fonctions primitives récursives : définition [transp. 16]**

▶ **PR** est la plus petite classe de fonctions $\mathbb{N}^k \dashrightarrow \mathbb{N}$ (en fait $\mathbb{N}^k \to \mathbb{N}$), pour $k$ variable qui :

- ▶ contient les projections $\underline{x} := (x_1, \dots, x_k) \mapsto x_i$ ;
- ▶ contient les constantes $\underline{x} \mapsto c$ ;
- ▶ contient la fonction successeur $x \mapsto x + 1$ ;
- ▶ est stable par composition : si $g_1, \dots, g_\ell \colon \mathbb{N}^k \dashrightarrow \mathbb{N}$ et $h \colon \mathbb{N}^\ell \dashrightarrow \mathbb{N}$ sont p.r. alors $\underline{x} \mapsto h(g_1(\underline{x}), \dots, g_\ell(\underline{x}))$ est p.r. ;
- ▶ est stable par récursion primitive : si $g \colon \mathbb{N}^k \dashrightarrow \mathbb{N}$ et $h \colon \mathbb{N}^{k+2} \dashrightarrow \mathbb{N}$ sont p.r., alors $f \colon \mathbb{N}^{k+1} \dashrightarrow \mathbb{N}$ est p.r., où :
$$f(\underline{x}, 0) = g(\underline{x})$$
$$f(\underline{x}, z+1) = h(\underline{x}, f(\underline{x}, z), z)$$

Les fonctions p.r. sont automatiq$^{\text{t}}$ totales, mais il est commode de garder la définition avec $\dashrightarrow$.

**Fonctions primitives récursives : exemples [transp. 17]**

▶ $f \colon (x, z) \mapsto x + z$ est p.r. :
$$f(x, 0) = x$$
$$f(x, z+1) = f(x, z) + 1$$
où $x \mapsto x$ et $(x, y, z) \mapsto y + 1$ sont p.r.

▶ $f \colon (x, z) \mapsto x \cdot z$ est p.r. :
$$f(x, 0) = 0$$
$$f(x, z+1) = f(x, z) + x$$

▶ $f \colon (x, z) \mapsto x^z$ est p.r.

▶ $f : (x, y, 0) \mapsto x,\ (x, y, z) \mapsto y$ si $z \geq 1$ est p.r. :

$$f(x, y, 0) = x$$
$$f(x, y, z + 1) = y$$

▶ $u \mapsto \max(u - 1, 0)$ est p.r. (exercice !), comme $(u, v) \mapsto \max(u - v, 0)$ ou $(u, v) \mapsto u\%v$ ou $(u, v) \mapsto \lfloor u/v \rfloor$.

**Fonctions primitives récursives : programmation** [transp. 18]

Les fonctions p.r. sont celles définies par un **langage de programmation à boucles bornées**, c'est-à-dire que :

- ▶ les variables sont des entiers naturels (illimités !),
- ▶ les manipulations de base sont permises (constantes, affectations, test d'égalité, conditionnelles),
- ▶ les opérations arithmétiques basiques sont disponibles,
- ▶ on peut faire des appels de fonctions *sans appels récursifs*,
- ▶ on ne peut faire que des boucles *de nombre borné a priori* d'itérations.

Les programmes dans un tel langage **terminent forcément par construction**.

**N.B.** $(m, n) \mapsto \langle m, n \rangle := m + \frac{1}{2}(m+n)(m+n+1)$ et $\langle m, n \rangle \mapsto m$ et $\langle m, n \rangle \mapsto n$ sont p.r.

**Fonctions primitives récursives : lien avec la complexité** [transp. 19]

En anticipant sur la notion de machine de Turing :

▶ La fonction $(M, C) \mapsto C'$ qui à une machine de Turing $M$ et une configuration (= ruban+état) $C$ de $M$ associe la configuration suivante **est p.r.**

▶ Conséquence : la fonction $(n, M, C) \mapsto C^{(n)}$ qui à $n \in \mathbb{N}$ et une machine de Turing $M$ et une configuration $C$ de $M$ associe la configuration atteinte après $n$ étapes d'exécution, **est p.r.**
(Par récursion primitive sur le point précédent.)

▶ Conséquence : une fonction calculable en complexité p.r. par une machine de Turing est elle-même p.r.
(Calculer une borne p.r. sur le nombre d'étapes, puis appliquer le point précédent.)

▶ Réciproquement : une p.r. est calculable en complexité p.r.

▶ Moralité : p.r. $\Leftrightarrow$ de complexité p.r.
Notamment **EXPTIME** $\subseteq$ **PR**.

### Fonctions primitives récursives : limitations [transp. 20]

La classe **PR** est « à cheval » entre la calculabilité et la complexité.

Rappel : la **fonction d'Ackermann** (pour $m = 2$) définie par :

$$A(2, n, 0) = 2 + n$$
$$A(2, 1, k + 1) = 2$$
$$A(2, n + 1, k + 1) = A(2,\ A(2, n, k + 1),\ k)$$

devrait être calculable. Mais cette définition *n'est pas une récursion primitive* (pourquoi ?).

▶ On peut montrer que : si $f\colon \mathbb{N}^k \to \mathbb{N}$ est p.r., il existe $r$ $(= r(f))$ tel que

$$f(x_1, \ldots, x_k) \leq A(2,\ (x_1 + \cdots + x_k + 3),\ r)$$

▶ Notamment, $r \mapsto A(2, r, r)$ **n'est pas p.r.**

Pourtant, *elle est bien définie par un algorithme* clair (et terminant clairement).

### Fonctions primitives récursives : numérotation (idée) [transp. 21]

▶ On veut *coder* les fonctions p.r. (et plus tard : gén$^{\text{ales}}$ récursives) *par des entiers.*

▶ Pour (certains) entiers $e \in \mathbb{N}$, on va définir $\psi_e^{(k)}\colon \mathbb{N}^k \to \mathbb{N}$ primitive récursive, la fonction p.r. *codée* par $e$ ou ayant $e$ comme **code** (source) / « programme ».

▶ Toute fonction p.r. $f\colon \mathbb{N}^k \to \mathbb{N}$ sera un $\psi_e^{(k)}$ pour un certain $e$.

▶ Ce $e$ décrit la manière dont $f$ est construite selon la définition de **PR** (cf. transp. 16).

▶ Il faut l'imaginer comme le *code source* de $f$ (au sens informatique).

▶ Il n'est *pas du tout unique* : $f = \psi_{e_1}^{(k)} = \psi_{e_2}^{(k)} = \cdots$
($e$ = « intention » / $f$ = « extension »)

▶ On va ensuite se demander si $(e, \underline{x}) \mapsto \psi_e^{(k)}(\underline{x})$ est *elle-même p.r.* (divulgâchis : *non*).

### Fonctions primitives récursives : numérotation (définition) [transp. 22]

On définit $\psi_e^{(k)}\colon \mathbb{N}^k \dashrightarrow \mathbb{N}$ par induction suivant la déf$^{\text{n}}$ de **PR** (cf. transp. 16) :

- ▶ si $e = \langle\langle 0, k, i\rangle\rangle$ alors $\psi_e^{(k)}(x_1 \ldots, x_k) = x_i$ (projections) ;
- ▶ si $e = \langle\langle 1, k, c\rangle\rangle$ alors $\psi_e^{(k)}(x_1 \ldots, x_k) = c$ (constantes) ;
- ▶ si $e = \langle\langle 2\rangle\rangle$ alors $\psi_e^{(1)}(x) = x + 1$ (successeur) ;
- ▶ si $e = \langle\langle 3, k, d, c_1, \ldots, c_\ell\rangle\rangle$ et $g_i := \psi_{c_i}^{(k)}$ et $h := \psi_d^{(\ell)}$, alors $\psi_e^{(k)}\colon \underline{x} \mapsto h(g_1(\underline{x}), \ldots, g_\ell(\underline{x}))$ (composition) ;
- ▶ si $e = \langle\langle 4, k, d, c\rangle\rangle$ et $g := \psi_c^{(k)}$ et $h := \psi_d^{(k+2)}$, alors (récursion primitive)

$$\psi_e^{(k+1)}(\underline{x}, 0) = g(\underline{x})$$
$$\psi_e^{(k+1)}(\underline{x}, z + 1) = h(\underline{x}, \psi_e^{(k+1)}(\underline{x}, z), z)$$

(Autres cas non définis, i.e., donnent ↑.)

▶ Alors $f\colon \mathbb{N}^k \dashrightarrow \mathbb{N}$ est p.r. *ssi* $\exists e \in \mathbb{N}.\, (f = \psi_e^{(k)})$ par définition.

P.ex., $e = \langle\langle 4, 1, \langle\langle 3, 3, \langle\langle 2\rangle\rangle, \langle\langle 0, 3, 2\rangle\rangle\rangle\rangle, \langle\langle 0, 1, 1\rangle\rangle\rangle\rangle = 1\,459\,411\,784\,487\ldots 780\,615\,609\,825 \approx 1.459 \times 10^{357}$ définit $\psi_e^{(2)}(x, z) = x + z$ avec les conventions de codage du transp. 11.

### Manipulation de programmes (version p.r.) [transp. 23]

▶ Penser à $e$ dans $\psi_e^{(k)}$ comme un programme écrit en « langage p.r. ».

▶ La fonction $\psi_e^{(k)} : \mathbb{N}^k \dashrightarrow \mathbb{N}$ « interprète » le programme $e$.

▶ Une fonction p.r. donnée a *beaucoup d'indices* : $\psi_{e_1}^{(k)} = \psi_{e_2}^{(k)} = \cdots$ (programmes équivalents).

*

La numérotation (transp. précédent) rend p.r. beaucoup de manipulations usuelles de programmes (composition, récursion, etc.). Notamment :

▶ **Théorème s-m-n** (Kleene) : il existe $s_{m,n} : \mathbb{N}^{m+1} \to \mathbb{N}$ p.r. telle que

$$(\forall e, \underline{x}, \underline{y}) \quad \psi^{(n)}_{s_{m,n}(e,x_1,\ldots,x_m)}(y_1,\ldots,y_n) = \psi_e^{(m+n)}(x_1,\ldots,x_m,\, y_1,\ldots,y_n)$$

Preuve : $s_{m,n}(e,\underline{x}) = \langle\!\langle 3, n, e, \langle\!\langle 1, n, x_1\rangle\!\rangle, \ldots, \langle\!\langle 1, n, x_m\rangle\!\rangle,\ \langle\!\langle 0, n, 1\rangle\!\rangle, \ldots, \langle\!\langle 0, n, n\rangle\!\rangle\rangle\!\rangle$ avec nos conventions (composition de fonctions constantes et de projections). □

*En clair :* $s_{m,n}$ prend un programme $e$ qui prend $m+n$ arguments en entrée et « fixe » la valeur des $m$ premiers arguments à $x_1, \ldots, x_m$, les $n$ arguments suivants $(y_1, \ldots, y_n)$ étant gardés variables.

### Digression : l'astuce de Quine (intuition) [transp. 24]

Le nom de Willard Van Orman Quine (1908–2000) a été associé à cette astuce par Douglas Hofstadter. En fait, l'astuce est plutôt due à Cantor, Gödel, Turing ou Kleene.

Les mots suivants suivis des mêmes mots entre guillemets forment une phrase intéressante : « les mots suivants suivis des mêmes mots entre guillemets forment une phrase intéressante ».

Pseudocode :

```
str="somefunc(code) { /*...*/ }\nsomefunc(\"str=\"+quote(str)+str);\n"; somefunc(code)
{ /*...*/ } somefunc("str="+quote(str)+str);
```

⇒ La fonction `somefunc` (arbitraire) est appelée avec le code source du programme *tout entier*.

**Exercice :** utiliser cette astuce pour écrire un programme écrivant son propre code source.

**Moralité :** *on peut toujours donner aux programmes accès à leur code source*, même si ce n'est pas prévu par le langage.

### Le théorème de récursion de Kleene (version p.r.) [transp. 25]

Version formelle de l'astuce de Quine

▶ **Théorème** (Kleene) : si $h : \mathbb{N}^{k+1} \dashrightarrow \mathbb{N}$ est p.r., il existe $e$ tel que

$$(\forall \underline{x}) \quad \psi_e^{(k)}(\underline{x}) = h(e, \underline{x})$$

Plus précisément, il existe $b : \mathbb{N}^2 \to \mathbb{N}$ p.r. telle que $e := b(k, d)$ vérifie

$$(\forall d)\,(\forall \underline{x}) \quad \psi_e^{(k)}(\underline{x}) = \psi_d^{(k+1)}(e, \underline{x})$$

Preuve : soit $s := s_{1,k}$ donné par le théorème s-m-n. La fonction $(t, \underline{x}) \mapsto h(s(t,t), \underline{x})$ est p.r., disons $= \psi_c^{(k+1)}(t, \underline{x})$. Alors

$$\psi^{(k)}_{s(c,c)}(\underline{x}) = \psi_c^{(k+1)}(c, \underline{x}) = h(s(c,c), \underline{x})$$

donc $e := s(c, c)$ convient. Les fonctions $d \mapsto c$ et $c \mapsto e$ sont p.r. □

**Moralité :** *on peut donner aux programmes accès à leur propre numéro* (= « code source », ici $e$), cela ne change rien.

**Fonctions primitives récursives : absence d'universalité [transp. 26]**

▶ **Théorème :** il n'existe pas de fonction p.r. $u\colon \mathbb{N}^2 \to \mathbb{N}$ telle que $u(e, x) = \psi_e^{(1)}(x)$ si $\psi_e^{(1)}(x) \downarrow$.

Preuve : par l'absurde : si un tel $u$ existe, alors $(e, x) \mapsto u(e, x) + 1$ est p.r. Par le théorème de récursion de Kleene, il existe $e$ tel que $\psi_e^{(1)}(x) = u(e, x) + 1$, ce qui contredit $u(e, x) = \psi_e^{(1)}(x)$. □

*

**Moralité :** *un interpréteur du langage p.r. ne peut pas être p.r.* (preuve : on peut interpréter l'interpréteur s'interprétant lui-même, en ajoutant 1 au résultat ceci donne un paradoxe ; c'est un argument diagonal de Cantor).

▶ Cet argument dépend du théorème s-m-n et du fait que les fonctions p.r. sont *totales*. Pour définir une théorie satisfaisante de la calculabilité, on va sacrifier la totalité pour sauver le théorème s-m-n.
Cette même preuve deviendra alors la preuve de l'indécidabilité du problème de l'arrêt.

**Fonctions primitives récursives : absence d'universalité (variante) [transp. 27]**

Rappel : la **fonction d'Ackermann** est définie par :

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

▶ Pour un $k$ *fixé*, la fonction $(m, n) \mapsto A(m, n, k)$ est p.r. (par récurrence sur $k$, récursion primitive sur $A(m, n, k - 1)$).

▶ Il existe même $k \mapsto a(k)$ p.r. telle que $\psi_{a(k)}^{(2)}(m, n) = A(m, n, k)$.

I.e., on peut calculer de façon p.r. en $k$ le *code* d'un programme p.r. qui calcule $(m, n) \mapsto A(m, n, k)$.

▶ Si (une extension de) $(e, n) \mapsto \psi_e^{(1)}(n)$ était p.r., on pourrait calculer $(n, k) \mapsto \psi_{s_{1,1}(a(k),2)}^{(1)}(n) = \psi_{a(k)}^{(2)}(2, n) = A(2, n, k)$, or elle n'est pas p.r.

# 3 Fonctions générales récursives

**Fonctions générales récursives : aperçu [transp. 28]**

▶ On a vu que les fonctions p.r. sont *limitées* et ne couvrent pas la notion générale d'algorithme :

- ▶ les algorithmes p.r. terminent toujours car
- ▶ le langage ne permet pas de boucles non bornées ;
- ▶ concrètement, il n'implémente pas la fonction d'Ackermann ;
- ▶ il ne peut pas s'interpréter lui-même.

▶ On veut modifier la définition des fonctions p.r. pour lever ces limitations. On va *autoriser les boucles infinies*.
→ Fonctions **générales récursives** ou simplement « **récursives** ».
Ce seront aussi nos fonctions **calculables** !

▶ En ce faisant, on obtient forcément des cas de non-terminaisons, donc on doit passer par des *fonctions partielles*.

**N.B.** Terminologie fluctuante : fonctions « générales récursives » ? juste « récursives » ? « récursives partielles » ? « calculables » ? « calculables partielles » ?

### L'opérateur $\mu$ de Kleene [transp. 29]

**Définition :** si $g\colon \mathbb{N}^{k+1} \dashrightarrow \mathbb{N}$ et $\underline{x} \in \mathbb{N}^k$, alors $\mu g(\underline{x})$ est le plus petit $z$ tel que $g(z, \underline{x}) = 0$ et $g(i, \underline{x}) \downarrow$ pour $0 \leq i < z$, s'il existe.

Autrement dit, $\mu g(\underline{x}) = z$ signifie

- ▶ $g(z, \underline{x}) = 0$,
- ▶ $g(i, \underline{x}) > 0$ (sous-entendant $g(i, \underline{x}) \downarrow$) pour tout $0 \leq i < z$.

Concrètement, penser à $\mu g$ comme la fonction
`i=0; while (true) { if (g(i,x)==0) { return i; } i++; }`

▶ Ceci permet toute sorte de *recherche non bornée*.

▶ L'opérateur $\mu$ associe à une fonction $g\colon \mathbb{N}^{k+1} \dashrightarrow \mathbb{N}$ une fonction $\mu g\colon \mathbb{N}^k \dashrightarrow \mathbb{N}$.

### Fonctions générales récursives : définition [transp. 30]

▶ **R** est la plus petite classe de fonctions $\mathbb{N}^k \dashrightarrow \mathbb{N}$, pour $k$ variable qui :

- ▶ contient les projections $\underline{x} := (x_1, \ldots, x_k) \mapsto x_i$ ;
- ▶ contient les constantes $\underline{x} \mapsto c$ ;
- ▶ contient la fonction successeur $x \mapsto x + 1$ ;
- ▶ est stable par composition : si $g_1, \ldots, g_\ell\colon \mathbb{N}^k \dashrightarrow \mathbb{N}$ et $h\colon \mathbb{N}^\ell \dashrightarrow \mathbb{N}$ sont récursives alors $\underline{x} \mapsto h(g_1(\underline{x}), \ldots, g_\ell(\underline{x}))$ est récursive ;
- ▶ est stable par récursion primitive : si $g\colon \mathbb{N}^k \dashrightarrow \mathbb{N}$ et $h\colon \mathbb{N}^{k+2} \dashrightarrow \mathbb{N}$ sont récursives, alors $f\colon \mathbb{N}^{k+1} \dashrightarrow \mathbb{N}$ est récursive, où :
$$f(\underline{x}, 0) = g(\underline{x})$$
$$f(\underline{x}, z+1) = h(\underline{x}, f(\underline{x}, z), z)$$
- ▶ est stable par l'opérateur $\mu$ : si $g\colon \mathbb{N}^{k+1} \dashrightarrow \mathbb{N}$ est récursive, alors $\mu g\colon \mathbb{N}^k \dashrightarrow \mathbb{N}$ est récursive (← nouveau !).

Cette fois le langage *permet les boucles non bornées* !

### Fonctions générales récursives : numérotation (idée) [transp. 31]

▶ On veut *coder* les fonctions générales récursives *par des entiers*.
Exactement comme on l'a fait pour les fonctions p.r., on change juste la notation de $\psi$ en $\varphi$.

▶ Pour (certains) entiers $e \in \mathbb{N}$, on va définir $\varphi_e^{(k)} \colon \mathbb{N}^k \dashrightarrow \mathbb{N}$ générale récursive, celle *codée* par $e$ ou ayant $e$ comme **code** (source) / « programme ».

▶ Toute fonction récursive (partielle !) $f \colon \mathbb{N}^k \dashrightarrow \mathbb{N}$ sera un $\varphi_e^{(k)}$ pour un certain $e$.

▶ Ce $e$ décrit la manière dont $f$ est construite selon la définition de $\mathbf{R}$ (cf. transp. 30).

▶ Il faut l'imaginer comme le *code source* de $f$ (au sens informatique).

▶ Il n'est *pas du tout unique* : $f = \varphi_{e_1}^{(k)} = \varphi_{e_2}^{(k)} = \cdots$
($e$ = « intention » / $f$ = « extension »)

▶ On va ensuite se demander si $(e, \underline{x}) \mapsto \varphi_e^{(k)}(\underline{x})$ est *elle-même récursive* (divulgâchis : *oui*, *contrairement* au cas p.r.).

### Fonctions générales récursives : numérotation (définition) [transp. 32]

On définit $\varphi_e^{(k)} \colon \mathbb{N}^k \dashrightarrow \mathbb{N}$ par induction suivant la déf$^{\text{n}}$ de $\mathbf{R}$ (cf. transp. 30) :

- ▶ si $e = \langle\!\langle 0, k, i \rangle\!\rangle$ alors $\varphi_e^{(k)}(x_1 \ldots, x_k) = x_i$ (projections) ;
- ▶ si $e = \langle\!\langle 1, k, c \rangle\!\rangle$ alors $\varphi_e^{(k)}(x_1 \ldots, x_k) = c$ (constantes) ;
- ▶ si $e = \langle\!\langle 2 \rangle\!\rangle$ alors $\varphi_e^{(1)}(x) = x + 1$ (successeur) ;
- ▶ si $e = \langle\!\langle 3, k, d, c_1, \ldots, c_\ell \rangle\!\rangle$ et $g_i := \varphi_{c_i}^{(k)}$ et $h := \varphi_d^{(\ell)}$, alors $\varphi_e^{(k)} \colon \underline{x} \mapsto h(g_1(\underline{x}), \ldots, g_\ell(\underline{x}))$ (composition) ;
- ▶ si $e = \langle\!\langle 4, k, d, c \rangle\!\rangle$ et $g := \varphi_c^{(k)}$ et $h := \varphi_d^{(k+2)}$, alors (récursion primitive)
$$\varphi_e^{(k+1)}(\underline{x}, 0) = g(\underline{x})$$
$$\varphi_e^{(k+1)}(\underline{x}, z+1) = h(\underline{x}, f(\underline{x}, z), z)$$
- ▶ si $e = \langle\!\langle 5, k, c \rangle\!\rangle$ et $g := \varphi_c^{(k+1)}$, alors $\varphi_e^{(k)} = \mu g$.

(Autres cas non définis, i.e., donnent $\uparrow$.)

▶ Alors $f \colon \mathbb{N}^k \dashrightarrow \mathbb{N}$ est récursive *ssi* $\exists e \in \mathbb{N}.\, (f = \varphi_e^{(k)})$ par définition.

### Fonctions générales récursives : universalité [transp. 33]

Cette fois, *il existe bien* une fonction récursive universelle, c'est-à-dire :

▶ $(e, \underline{x}) \mapsto \varphi_e^{(k)}(\underline{x})$ est récursive : $\exists u_k \in \mathbb{N}.\, (\varphi_{u_k}^{(k+1)}(e, \underline{x}) = \varphi_e^{(k)}(\underline{x}))$.
Variante : $\exists u.\, (\varphi_u^{(1)}(\langle\!\langle e, \underline{x} \rangle\!\rangle) = \varphi_e^{(k)}(\underline{x}))$ en codant programme $e$ et arguments $\underline{x}$ en un entier.

Qu'est-ce que ça nous dit ?

▶ Mathématiquement, que $\varphi^{(k)}$ est elle-même récursive en tous ses arguments, y compris l'indice de numérotation.

▶ Informatiquement, qu'il existe un *interpréteur* $u$ du langage général récursif dans le langage général récursif.

▶ Philosophiquement, que la notion d'« exécuter un algorithme » est *elle-même algorithmique*.

Mais comment le prouver ? On va esquisser une méthode par **arbres de calcul**.

### Arbres de calcul pour les fonctions récursives [transp. 34]

Un **arbre de calcul** de $\varphi_e^{(k)}(\underline{x})$ de résultat $y$ est un arbre (fini, enraciné, ordonné, étiqueté par des entiers naturels) qui « atteste » que $\varphi_e^{(k)}(\underline{x}) = y$ en détaillant les étapes du calcul.

- ▶ La racine est étiquetée « $\varphi_e^{(k)}(\underline{x}) = y$ » codé disons « $\langle\langle e, \langle\langle \underline{x} \rangle\rangle, y\rangle\rangle$ ».
- ▶ Le sous-arbre porté par chaque fils de la racine est lui-même un arbre de calcul pour une sous-expression utilisée dans le calcul de $f(\underline{x}) = y$.
- ▶ Pour les cas projection, constante, successeur, il n'y a pas de fils.
- ▶ Pour la composition $h(g_1(\underline{x}), \ldots, g_\ell(\underline{x}))$, les fils portent des arbres de calcul attestant $g_1(\underline{x}) = v_1, \ldots, g_\ell(\underline{x}) = v_\ell$ et $h(\underline{v}) = y$ (donc $\ell + 1$ fils).
- ▶ Pour la récursion primitive $f(\underline{x}, z)$, on a soit un seul fils attestant $g(\underline{x}) = y$ lorsque $z = 0$ soit deux fils attestant $f(\underline{x}, z') = v$ et $h(\underline{x}, v, z') = y$ lorsque $z = z' + 1$ (donc 1 ou 2 fils).
- ▶ Pour $f = \mu g$, on a des fils attestant $g(0, \underline{x}) = v_0, \ldots, g(y, \underline{x}) = v_y$ où $v_i > 0$ si $0 \leq i < y$ et $v_y = 0$ (donc $y + 1$ fils).

On a $\varphi_e^{(k)}(\underline{x}) = y$ *ssi* il en existe un arbre de calcul qui l'atteste.

### Arbres de calcul : formalisation [transp. 35]

Formellement :

- ▶ la racine est étiquetée $\langle\langle e, \langle\langle \underline{x} \rangle\rangle, y\rangle\rangle$,
- ▶ soit $e$ est $\langle\langle 0, k, i\rangle\rangle$ ou $\langle\langle 1, k, c\rangle\rangle$ ou $\langle\langle 2\rangle\rangle$, elle n'a pas de fils, et $y$ vaut resp$^{\text{t}}$ $x_i$, $c$ ou $x+1$,
- ▶ soit $e = \langle\langle 3, k, d, c_1, \ldots, c_\ell\rangle\rangle$ et les $\ell + 1$ fils portent des arbres de calculs attestant $\varphi_{c_1}^{(k)}(\underline{x}) = v_1$, ..., $\varphi_{c_\ell}^{(k)}(\underline{x}) = v_\ell$ et $\varphi_d^{(\ell)}(\underline{v}) = y$.
- ▶ soit $e = \langle\langle 4, k', d, c\rangle\rangle$ où $k' = k - 1$ et
  - — soit $x_k = 0$ et l'unique fils porte arbre de calcul attestant $\varphi_c^{(k')}(x_1, \ldots, x_{k'}) = y$,
  - — soit $x_k = z+1$ et les deux fils portent arbres de calcul attestant $\varphi_e^{(k'+1)}(x_1, \ldots, x_{k'}, z) = v$ et $\varphi_d^{(k'+2)}(x_1, \ldots, x_{k'}, v, z) = y$.
- ▶ soit $e = \langle\langle 5, k, c\rangle\rangle$ et les $y+1$ fils portent des arbres de calcul attestant $\varphi_c^{(k+1)}(0, x_1, \ldots, x_k) = v_0$, ..., $\varphi_c^{(k+1)}(y, x_1, \ldots, x_k) = v_y$ où $v_y = 0$ et $v_0, \ldots, v_{y-1} > 0$.

On encode l'arbre $\mathscr{T}$ par l'entier $\text{code}(\mathscr{T}) := \langle\langle n, \text{code}(\mathscr{T}_1), \ldots, \text{code}(\mathscr{T}_s)\rangle\rangle$ où $n$ est l'étiquette de la racine et $\mathscr{T}_1, \ldots, \mathscr{T}_s$ les codes des sous-arbres portés par les $s$ fils de celle-ci.

### Arbres de calcul ⇒ universalité [transp. 36]

Les points-clés :

- ▶ On a $\varphi_e^{(k)}(\underline{x}) = y$ *ssi* il existe un arbre de calcul $\mathscr{T}$ l'attestant.

- Vérifier si $\mathscr{T}$ est un arbre de calcul valable est *primitif récursif* en code($\mathscr{T}$). (On peut vérifier les règles à chaque nœud avec des boucles bornées.)
- De même, extraire $e, \underline{x}, y$ de $\mathscr{T}$ est primitif récursif.

D'où l'algorithme « universel » pour calculer $\varphi_e^{(k)}(\underline{x})$ en fonction de $e, \underline{x}$ :

- parcourir $n = 0, 1, 2, 3, 4, \ldots$ (boucle non bornée),
- pour chacun, tester s'il code un arbre de calcul valable de $\varphi_e^{(k)}(\underline{x})$,
- si oui, terminer et renvoyer le $y$ contenu.

La boucle non-bornée est précisément ce que permet $\mu$. Tout le reste est p.r.
$\Rightarrow$ Ceci montre l'existence de $u$ (code de l'algorithme décrit ci-dessus).

Ne pas coder un interpréteur comme ça dans la vraie vie !

**Théorème de la forme normale** **[transp. 37]**

On a montré un peu plus que l'universalité : on peut exécuter n'importe quel algorithme avec une *unique boucle non bornée*. Plus exactement :

- **Théorème de la forme normale** (Kleene) : il existe un prédicat p.r. $T$ sur $\mathbb{N}^3$ et une fonction p.r. $U : \mathbb{N} \to \mathbb{N}$ tels que :

$$\varphi_e^{(k)}(x_1, \ldots, x_k) = U(\mu T(e, \langle\!\langle x_1, \ldots, x_k \rangle\!\rangle))$$

Précisément, $T(n, e, \langle\!\langle x_1, \ldots, x_k \rangle\!\rangle)$ teste si $n$ est le code d'un arbre de calcul valable de $\varphi_e^{(k)}(\underline{x})$, et $U$ extrait le résultat de cet arbre.

*

Exemple d'application : **lancement en parallèle** :

$$U(\mu(T(e_1, \langle\!\langle \underline{x} \rangle\!\rangle) \text{ ou } T(e_2, \langle\!\langle \underline{x} \rangle\!\rangle)))$$

définit (de façon p.r. en $e_1, e_2$) un $e$ tel que

$$\varphi_e(\underline{x})\downarrow \iff \varphi_{e_1}(\underline{x})\downarrow \text{ ou } \varphi_{e_2}(\underline{x})\downarrow$$

**Théorème s-m-n (version générale récursive)** **[transp. 38]**

Exactement comme la version p.r. :

- **Théorème s-m-n** (Kleene) : il existe $s_{m,n} : \mathbb{N}^{m+1} \to \mathbb{N}$ p.r. telle que

$$(\forall e, \underline{x}, \underline{y}) \quad \varphi^{(n)}_{s_{m,n}(e,x_1,\ldots,x_m)}(y_1, \ldots, y_n) = \varphi_e^{(m+n)}(x_1, \ldots, x_m, y_1, \ldots, y_n)$$

Noter que $s_{m,n}$ est *primitive récursive* même si on s'intéresse ici aux fonctions générales récursives.

Les manipulations de programmes sont typiquement p.r. (même si les programmes manipulés sont des fonctions générales récursives).

### Arité et encodage des tuples [transp. 39]

Remarque qui aurait dû être faite avant ?

Pour tout $k \geq q$, les fonctions

$$\begin{cases} \mathbb{N}^k \to \mathbb{N} \\ (x_1, \ldots, x_k) \mapsto \langle\langle x_1, \ldots, x_k \rangle\rangle \end{cases} \quad \text{et} \quad \begin{cases} \mathbb{N} \to \mathbb{N} \\ \langle\langle x_1, \ldots, x_k \rangle\rangle \mapsto x_i \end{cases}$$

sont p.r. Par conséquent,

$$f \colon \mathbb{N}^k \dashrightarrow \mathbb{N} \text{ récursive} \iff \begin{cases} \mathring{f} \colon \mathbb{N} \dashrightarrow \mathbb{N} \\ \langle\langle x_1, \ldots, x_k \rangle\rangle \mapsto f(x_1, \ldots, x_k) \end{cases} \text{ récursive}$$

et de plus, un numéro $e$ de $f$ (i.e., $f = \varphi_e^{(k)}$) se calcule de façon p.r. à partir d'un numéro $e'$ de $\mathring{f}$ (i.e., $\mathring{f} = \varphi_{e'}^{(1)}$) et vice versa.

Ceci justifie d'omettre parfois abusivement l'arité
(par défaut, « $\varphi_e$ » désigne « $\varphi_e^{(1)}$ »).

Même chose, *mutatis mutandis* (avec $\psi$) pour les fonctions p.r. elles-mêmes.

### Le théorème de récursion de Kleene (version générale récursive) [transp. 40]

Exactement comme la version p.r. :

▶ **Théorème** (Kleene) : si $h \colon \mathbb{N}^{k+1} \dashrightarrow \mathbb{N}$ est récursive, il existe $e$ tel que

$$(\forall \underline{x}) \quad \varphi_e^{(k)}(\underline{x}) = h(e, \underline{x})$$

Plus précisément, il existe $b \colon \mathbb{N}^2 \to \mathbb{N}$ p.r. telle que $e := b(k, d)$ vérifie

$$(\forall d)\,(\forall \underline{x}) \quad \varphi_e^{(k)}(\underline{x}) = \varphi_d^{(k+1)}(e, \underline{x})$$

Même preuve : soit $s := s_{1,k}$ donné par le théorème s-m-n. La fonction $(t, \underline{x}) \mapsto h(s(t,t), \underline{x})$ est p.r., disons $= \varphi_c^{(k+1)}(t, \underline{x})$. Alors

$$\varphi_{s(c,c)}^{(k)}(\underline{x}) = \varphi_c^{(k+1)}(c, \underline{x}) = h(s(c,c), \underline{x})$$

donc $e := s(c,c)$ convient. Les fonctions $d \mapsto c$ et $c \mapsto e$ sont p.r. □

**Moralité :** *on peut donner aux programmes accès à leur propre numéro* (= « code source »), cela ne change rien.

### Le théorème du point fixe de Kleene-Rogers [transp. 41]

Reformulation du théorème de récursion utilisant l'universalité :

▶ **Théorème** (Kleene-Rogers) : si $F \colon \mathbb{N} \dashrightarrow \mathbb{N}$ est récursive et $k \in \mathbb{N}$, il existe $e$ tel que

$$\varphi_e^{(k)} = \varphi_{F(e)}^{(k)}$$

Preuve : $h \colon (e, \underline{x}) \mapsto \varphi_{F(e)}^{(k)}(\underline{x})$ est récursive car $e \mapsto F(e)$ l'est et que $(e', \underline{x}) \mapsto \varphi_{e'}^{(k)}(\underline{x})$ l'est (universalité). Par le théorème de récursion, il existe $e$ tel que $\varphi_e^{(k)}(\underline{x}) = h(e, \underline{x}) = \varphi_{F(e)}^{(k)}(\underline{x})$. □

**Moralité :** quelle que soit la transformation $F$ calculable effectuée sur le source d'un programme, il y a un programme $e$ qui fait la même chose que son transformé $F(e)$.

### Récursion ! [transp. 42]

Le langage des fonctions générales récursives, malgré le nom ne permet pas les définitions par appels récursifs.

Uniquement des opérations élémentaires, appels de fonctions précédemment définies, boucles.

Comment permettre quand même les appels récursifs ?

*Par le théorème de récursion de Kleene !* (ou théorème du point fixe) :

- je veux définir (comme fonction générale récursive) une fonction $f$ dont la définition fait appel à $f$ elle-même :
- par le théorème de récursion de Kleene (« astuce de Quine »), je peux supposer que $f$ a accès à son propre numéro (« code source »),
- je convertis chaque appel à $f$ depuis $f$ en un appel à la fonction universelle (interpréteur) sur le numéro de $f$.

Ne pas implémenter la récursion comme ça dans la vraie vie !

### *Kids, don't try this at home !* [transp. 43]

Pseudocode :

```
fibonacci(n) { str = "self = \"fibonacci(n) {\\nstr = \" + quote(str) + str;\n\ if
(n==0 || n==1) return n;\n\ return interpret(self, n-1) + interpret(self, n-2);\n\
}"; self = "fibonacci(n) {\nstr = " + quote(str) + str; if (n==0 || n==1) return n;
return interpret(self, n-1) + interpret(self, n-2); }
```

*

**Défi :** trouver explicitement un $e$ tel que $\varphi_e^{(3)}$ soit la fonction d'Ackermann.

(La fonction d'Ackermann a été définie par des appels récursifs donc elle est bien censée être calculable.)

### Le problème de l'arrêt [transp. 44]

Le terme « problème de l'arrêt » prendra plus de sens pour les machines de Turing.

▶ **Problème :** donné un programme $e$ (mettons d'arité $k = 1$) et une entrée $x$ à ce programme, comment savoir si l'algorithme $e$ termine (c'est-à-dire $\varphi_e^{(1)}(x) \downarrow$) ou non ($\varphi_e^{(1)}(x) \uparrow$) sur cette entrée ?

Cette question est-elle *algorithmique* ?

**Réponse** de Turing : *non.*

▶ Intuition de la preuve : supposons que j'aie un moyen algorithmique pour savoir si un algorithme termine ou pas, je peux lui demander ce que « je » vais faire (astuce de Quine !), et faire le contraire, ce qui conduit à un paradoxe.

### L'indécidabilité du problème de l'arrêt [transp. 45]

► **Théorème** (Turing) : il n'existe pas de fonction récursive $h\colon \mathbb{N}^2 \to \mathbb{N}$ telle que $h(e,x) = 1$ si $\varphi_e^{(1)}(x) \downarrow$ et $h(e,x) = 0$ si $\varphi_e^{(1)}(x) \uparrow$.

Preuve : par l'absurde : si un tel $h$ existe, alors la fonction

$$v\colon (e,x) \mapsto \begin{cases} 42 & \text{si } h(e,x) = 0 \\ \uparrow & \text{si } h(e,x) = 1 \end{cases}$$

est générale récursive (tester is $h(e,x) = 0$, si oui renvoyer 42, sinon faire une boucle infinie, p.ex. $\mu(x \mapsto 1)$).

Par le théorème de récursion de Kleene, il existe $e$ tel que $\varphi_e^{(1)}(x) = v(e,x)$.

Si $\varphi_e^{(1)}(x) \downarrow$ alors $h(e,x) = 1$ donc $v(e,x) \uparrow$ donc $\varphi_e^{(1)}(x) \uparrow$, contradiction.
Si $\varphi_e^{(1)}(x) \uparrow$ alors $h(e,x) = 0$ donc $v(e,x) \downarrow$ donc $\varphi_e^{(1)}(x) \downarrow$, contradiction. □

### L'indécidabilité du problème de l'arrêt : redite [transp. 46]

Notons $\varphi$ pour $\varphi^{(1)}$.

► **Théorème** (Turing) : il n'existe pas de fonction récursive $h\colon \mathbb{N}^2 \to \mathbb{N}$ telle que $h(e,x) = 1$ si $\varphi_e(x) \downarrow$ et $h(e,x) = 0$ si $\varphi_e(x) \uparrow$.

Preuve (incluant celle du théorème de récursion) : considérons la fonction $v\colon \mathbb{N} \dashrightarrow \mathbb{N}$ qui à $e$ associe 42 si $h(e,e) = 0$ et $\uparrow$ (non définie) si $h(e,e) = 1$.
Supposons par l'absurde $h$ est calculable : alors cette fonction (partielle) $v$ est calculable, disons $v = \varphi_c$.
Si $\varphi_c(c) \downarrow$ alors $h(c,c) = 1$ donc $v(c) \uparrow$, c'est-à-dire $\varphi_c(c) \uparrow$, contradiction. Si $\varphi_c(c) \uparrow$ alors $h(c,c) = 0$ donc $v(c) \downarrow$, c'est-à-dire $\varphi_c(c) \downarrow$, contradiction. □

C'est un *argument diagonal* : on utilise $h$ pour construire une fonction qui diffère en tout point de la diagonale $c \mapsto \varphi_c(c)$, donc elle ne peut pas être une $\varphi_c$.

Pour les fonctions p.r. (qui terminent toujours !), le même argument diagonal donnait l'inexistence d'un programme universel (transp. 26).

### Comparaison fonctions primitives récursives et générales récursives [transp. 47]

Récapitulation :

► Les fonctions p.r. sont totales ; les générales récursives sont possiblement partielles.

► Les fonctions p.r. sont un langage limité (pas de boucle non bornées a priori) ; les générales récursives coïncideront avec les fonctions « calculables » (équivalence avec machines de Turing et $\lambda$-calcul à voir).

► Les fonctions p.r. ne permettent pas d'interpréter les fonctions p.r. ; les générales récursives peuvent s'interpréter elles-mêmes (universalité) et donc réaliser n'importe quelle sorte d'appels récursifs.

► Le problème de l'arrêt pour les fonctions p.r. est trivial (elles sont totales !) ; pour les fonctions générales récursives, il est indécidable (= pas calculable par une fonction générale récursive).

# 4 Machines de Turing

**Machines de Turing : explication informelle [transp. 48]**

La **machine de Turing** est une modélisation d'un ordinateur extrêmement simple, réalisant des calculs indiscutablement finitistes.

C'est une sorte d'automate doté d'un **état** interne pouvant prendre un nombre fini de valeurs, et d'une mémoire illimitée sous forme de **bande** linéaire divisée en cellules (indéfiniment réécrivibles), chaque cellule pouvant contenir un **symbole**.

La machine peut observer, outre son état interne, une unique case de la bande, là où se trouve sa **tête de lecture/écriture**.

Le **programme** de la machine indique, pour chaque combinaison de l'état interne et du symbole lu par la tête :

- dans quel état passer,
- quel symbole écrire à la place de la tête,
- la direction dans laquelle déplacer la tête (gauche ou droite).

La machine suit son programme jusqu'à tomber dans un état spécial 0 (« arrêt »).

**Machines de Turing : définition [transp. 49]**

Une **machine de Turing** (déterministe) à (1 bande, 2 symboles et) $m \geq 2$ états est la donnée de :

- un ensemble fini $Q$ de cardinal $m$ d'**états**, qu'on identifiera à $\{0, \ldots, m-1\}$,
- un ensemble $\Sigma$ de (ici) 2 **symboles de bande** qu'on identifiera à $\{0, 1\}$,
- une fonction
$$\delta \colon (Q \setminus \{0\}) \times \Sigma \to Q \times \Sigma \times \{\mathtt{L}, \mathtt{R}\}$$
appelé **programme** de la machine.

(Il y a donc $(4m)^{2(m-1)}$ machines à $m$ états.)

Une **configuration** d'une telle machine est la donnée de :

- un élément $q \in Q$ appelé l'**état courant**,
- une fonction $\beta \colon \mathbb{Z} \to \Sigma$ ne prenant qu'*un nombre fini* de valeurs $\neq 0$, appelée la **bande**,
- un entier $i \in \mathbb{Z}$ appelé la **position de la tête** sur la bande.

**Machines de Turing : exécution d'une étape [transp. 50]**

Si $(q, \beta, i)$ est une configuration de la machine de Turing où $q \neq 0$, et $\delta$ le programme, la **configuration suivante** est $(q', \beta', i')$ où :

- $(q', y, d) = \delta(q, \beta(i))$ est l'**instruction exécutée**,
- $q'$ est le **nouvel état**,
- $i' = i - 1$ si $d = \mathtt{L}$ et $i' = i + 1$ si $d = \mathtt{R}$,
- $\beta'(j) = \beta(j)$ pour $j \neq i$ tandis que $\beta'(i) = y$.

*En clair :* le programme indique, pour chaque configuration d'un état $\neq 0$ et d'un symbole $x = \beta(i)$ lu sur la bande :

- le nouvel état $q'$ dans lequel passer,
- le symbole $y$ à écrire à la place de $x$ à l'emplacement $i$ de la bande,
- la direction dans laquelle déplacer la tête (gauche ou droite).

**Machines de Turing : exécution complète** [transp. 51]

▶ Si $C = (q, \beta, i)$ est une configuration d'une machine de Turing, la **trace d'exécution** à partir de $C$ est la suite finie ou infinie $C^{(0)}, C^{(1)}, C^{(2)}, \ldots$, où

- $C^{(0)} = C$ est la configuration donnée (configuration initiale),
- si $C^{(n)} = (q^{(n)}, \beta^{(n)}, i^{(n)})$ avec $q^{(n)} = 0$ alors la suite s'arrête ici, on dit que **la machine s'arrête**, que $C^{(n)}$ est la **configuration finale**, et que l'exécution a duré $n$ **étapes**,
- sinon, $C^{(n+1)}$ est la configuration suivante (définie transp. précédent).

*En clair :* la machine continue à exécuter des instructions tant qu'elle n'est pas tombée dans l'état 0. Elle s'arrête quand elle tombe dans l'état 0.

**Simulation des machines de Turing par les fonctions récursives** [transp. 52]

▶ On peut coder un programme et/ou une configuration sous forme d'entiers naturels.
Le ruban a un nombre *fini* de symboles $\neq 0$, donc on peut le coder par la liste de leurs positions comptées, disons, à partir du symbole $\neq 0$ le plus à gauche.

▶ La fonction $(M, C) \mapsto C'$ qui à une machine de Turing $M$ et une configuration $C$ de $M$ associe la configuration suivante **est p.r.**

▶ Conséquence : la fonction $(n, M, C) \mapsto C^{(n)}$ qui à $n \in \mathbb{N}$ et une machine de Turing $M$ et une configuration $C$ de $M$ associe la configuration atteinte après $n$ étapes d'exécution, **est p.r.**

▶ La fonction qui à $(M, C)$ associe la configuration finale (et/ou le nombre d'étapes d'exécution) *si la machine s'arrête*, et $\uparrow$ (non définie) si elle ne s'arrête pas, est **générale récursive**.

**Moralité :** les fonctions récursives peuvent simuler les machines de Turing.

**Calculs sur machines de Turing : une convention** [transp. 53]

On dira qu'une fonction $f : \mathbb{N}^k \dashrightarrow \mathbb{N}$ est **calculable par machine de Turing** lorsqu'il existe une machine de Turing qui, pour tous $x_1, \ldots, x_k$ :

- part de la configuration initiale suivante : l'état est 1, les symboles $\beta(j)$ du ruban pour $j < 0$ sont arbitraires (tous 0 sauf un nombre fini), la tête est à l'emplacement 0,
- les symboles $\beta(j)$ pour $j \geq 0$ du ruban initial forment le mot suivant :
$$01^{x_1}01^{x_2}0\cdots01^{x_k}0$$
(suivi d'une infinité de 0), c'est-à-dire $\beta(0) = 0$, $\beta(j) = 1$ si $1 \leq j \leq x_1$, $\beta(1 + x_1) = 0$, $\beta(j) = 1$ si $2 + x_1 \leq j \leq 1 + x_1 + x_2$, etc.,

- si $f(x_1, \ldots, x_k) \uparrow$, la machine ne s'arrête pas,
- si $f(x_1, \ldots, x_k)\downarrow = y$, la machine s'arrête avec la tête à l'emplacement 0 (le même qu'au départ), le ruban $\beta(j)$ non modifié pour $j < 0$, et
- les symboles $\beta(j)$ pour $j \geq 0$ du ruban final forment le mot $01^y0$ (suivi d'une infinité de 0) (« codage unaire » de $y$).

**Calculs par les machines de Turing des fonctions récursives [transp. 54]**

▶ On peut montrer par induction suivant la déf$^{\text{n}}$ de $\mathbf{R}$ que *toute fonction générale récursive est calculable par machine de Turing* avec les conventions du transp. précédent.

▶ La démonstration est fastidieuse mais pas difficile : il s'agit essentiellement de programmer en machine de Turing chacune des formes de construction des fonctions générales récursives (projections, constantes, successeur, composition, récursion primitive, $\mu$-récursion).

▶ Les conventions faites, notamment le fait d'ignorer et de ne pas modifier $\beta(j)$ pour $j < 0$, permettent à l'induction dans la preuve de fonctionner.

Par exemple, pour la composition, on va utiliser cette propriété pour « sauvegarder » les $x_1, \ldots, x_k$ initiaux, ainsi que les valeurs de $g_j(\underline{x})$ calculées, lorsqu'on appelle chacune des fonctions $g_1, \ldots, g_\ell$ (à chaque fois, on les recopie $x_1, \ldots, x_k$ à droite des valeurs à ne pas toucher, et on appelle la machine calculant $g_j$ sur ces valeurs recopiées).

**Équivalence entre machines de Turing et fonctions récursives [transp. 55]**

▶ Toute fonction générale récursive $\mathbb{N}^k \dashrightarrow \mathbb{N}$ est calculable par machine de Turing (sous les conventions données) : transp. précédent.

▶ Réciproquement, toute fonction $\mathbb{N}^k \dashrightarrow \mathbb{N}$ calculable par machine de Turing sous ces conventions est générale récursive, car les fonctions récursives peuvent simuler les machines de Turing, calculer une configuration initiale convenable, et décoder la configuration finale (cf. transp. 52).

▶ Bref, $f\colon \mathbb{N}^k \dashrightarrow \mathbb{N}$ est calculable par machine de Turing *ssi* elle est générale récursive.

▶ De plus, cette équivalence est *constructive* : il existe des fonctions p.r. :
- l'une prend en entrée le numéro $e$ d'une fonction générale récursive (et l'arité $k$) et renvoie le code d'une machine de Turing qui calcule cette $\varphi_e^{(k)}$,
- l'autre prend en entrée le code d'une machine de Turing qui calcule une fonction $f$, et son arité $k$, et renvoie un numéro $e$ de $f$ dans les fonctions générales récursives $f = \varphi_e^{(k)}$.

### Machines de Turing : variations [transp. 56]

On a choisi ici une notion de machine de Turing assez restreinte (1 bande, 2 symboles de bande). Il existe toutes sortes de variations :

- machines à plusieurs bandes (mais en nombre fini ; le programme choisit en fonction du symbole lu sur chaque bande, et écrit et déplace chaque tête indépendamment), voire à plusieurs têtes par bande, parfois avec des bandes en lecture seule (pour les entrées), ou en écriture seule (pour les sorties),
- autres symboles que 0 et 1 (mais en nombre fini),
- machine non-déterministe (plusieurs instructions possibles dans une configuration donnée ; la machine termine si au moins l'un des chemins d'exécution termine).

Du point de vue *calculabilité*, ces modifications ne rendent pas la machine plus puissante, et, sauf, cas dégénérés (p.ex., un seul symbole sur le ruban !) elles ne la rendent pas moins puissante non plus. Ceci confirme la robustesse du modèle de Church-Turing.

Pour la *complexité*, en revanche, c'est une autre affaire.

### Machines de Turing : reprise de résultats déjà vus [transp. 57]

▶ **Universalité :** pour un codage raisonnable, il existe une machine de Turing « universelle » qui prend en entrée sur sa bande le programme d'une autre machine de Turing $M$, et une configuration initiale $C$ pour celle-ci, et qui simule l'exécution de $M$ sur $C$ (notamment, elle s'arrête ssi $M$ s'arrête).

▶ **Forme normale :** la fonction $(n, M, C) \mapsto C^{(n)}$ qui à $n \in \mathbb{N}$ et une machine de Turing $M$ et une configuration $C$ de $M$ associe la configuration après $n$ étapes d'exécution, est p.r., et en particulier, calculable par une machine de Turing.

⇒ En particulier, on peut tester algorithmiquement si une machine de Turing donnée, depuis une configuration initiale donnée, s'arrête *en moins de $n$ étapes*.

▶ **Indécidabilité du problème de l'arrêt :** la fonction qui à $(M, C)$ associe 1 si la machine de Turing s'arrête en partant de la configuration initiale $C$, et 0 sinon, *n'est pas calculable*.

⇒ On ne peut pas tester algorithmiquement si une machine de Turing donnée, depuis une configuration initiale donnée, s'arrête « un jour ».

### Indécidabilité du problème de l'arrêt (départ bande vierge) [transp. 58]

On ne peut même pas tester algorithmiquement si une machine de Turing s'arrête à partir d'une bande vierge :

▶ **Indécidabilité du problème de l'arrêt :** la fonction qui à $M$ associe 1 si la machine de Turing s'arrête en partant de la configuration vierge $C_0$ (c'est-à-dire celle où $\beta = 0$, état initial 1), et 0 sinon, *n'est pas calculable*.

Preuve : Supposons par l'absurde qu'on puisse tester algorithmiquement si une machine de Turing s'arrête à partir d'une configuration vierge. On va montrer qu'on peut tester si une machine de Turing $M$ s'arrête à partir de $C$ quelconque.

Données $M$ et $C$, on peut algorithmiquement calculer une machine $N$ qui « prépare » $C$ à partir de la configuration vierge $C_0$, donc une machine $M^*$ qui exécute successivement $N$ puis $M$ (← ceci est un théorème s-m-n).

Ainsi $M^*$ (calculé algorithmiquement) termine sur $C_0$ ssi $M$ termine sur $C$.

Donc tester la terminaison de $M^*$ permettrait de tester celle de $M$ sur $C$, ce qui n'est pas possible (← ceci est une preuve « par réduction »). □

### Le castor affairé [transp. 59]

▶ La fonction **castor affairé** associe à $m$ le nombre maximal $B(m)$ d'étapes d'exécution d'une machine de Turing à $\leq m$ états *qui termine* (à partir de la configuration vierge $C_0$).

▶ La fonction $B$ croît *trop vite pour être calculable* :

$$\forall f\colon \mathbb{N} \to \mathbb{N} \text{ calculable.} \quad \exists m \in \mathbb{N}. \quad (B(m) > f(m))$$

Preuve : supposons au contraire $\forall m \in \mathbb{N}.\ (B(m) \leq f(m))$ avec $f\colon \mathbb{N} \to \mathbb{N}$ calculable. Donnée une machine de Turing $M$, on peut alors algorithmiquement décider si $M$ s'arrête à partir de $C_0$ :

- ▶ calculer $f(m)$ où $m$ est le nombre d'états de $M$,
- ▶ exécuter $M$ à partir de $C_0$ pendant $f(m)$ étapes (ce nombre est $\geq B(m)$ par hypothèse),
- ▶ si elle a terminé en temps imparti, $M$ termine sur $C_0$, et on renvoie « oui » ; sinon, elle ne termine jamais par définition de $B(m)$, on renvoie « non ».

Ceci est impossible donc $f$ n'est pas calculable. □

### Le castor affairé (amélioration) [transp. 60]

$B(m)$ = nombre maximal d'étapes d'exécution d'une machine de Turing à $\leq m$ états *qui termine* à partir d'une bande vierge.

▶ On peut faire mieux : $B$ *domine* toute fonction calculable :

$$\forall f\colon \mathbb{N} \to \mathbb{N} \text{ calculable.} \quad \exists m_0 \in \mathbb{N}. \quad \forall m \geq m_0. \quad (B(m) > f(m))$$

Preuve : soit $f\colon \mathbb{N} \to \mathbb{N}$ calculable. Soit $\gamma(r) = A(2, r, r)$ (en fait, $2^r$ doit suffire ; noter $\gamma$ croissante). Pour $r \in \mathbb{N}$, on considère la machine de Turing $M_r$ qui

- ▶ prépare $r$, calcule $\gamma(r+1)$ puis $f(0) + f(1) + \cdots + f(\gamma(r+1)) + 1$,
- ▶ attend ce nombre-là d'étapes, et termine.

Le nombre d'états de $M_r$ est une fonction p.r. $b(r)$ de $r$ (même $b(r) = r + \text{const}$ convient). Pour $r \geq r_0$ on a $b(r) \leq \gamma(r)$. Soit $m_0 = \gamma(r_0)$. Si $m \geq m_0$, soit $r \geq r_0$ tel que $\gamma(r) \leq m \leq \gamma(r+1)$. Alors $M_r$ calcule $\cdots + f(m) + \cdots + 1$, donc attend $> f(m)$ étapes. Donc $B(b(r)) > f(m)$. Mais $b(r) \leq \gamma(r) \leq m$ donc $B(m) > f(m)$. □

*

▶ Variations du castor affairé : nombre de symboles écrits sur la bande, $n \mapsto \max\{\varphi_e(e) : 0 \leq e \leq n \text{ et } \varphi_e(e)\downarrow\}$ (mêmes propriétés).

# 5 Décidabilité et semi-décidabilité

**Terminologie calculable/décidable [transp. 61]**

▶ Une fonction partielle $f\colon \mathbb{N}^k \dashrightarrow \mathbb{N}$ est dite **calculable** (partielle) lorsqu'elle est (c'est équivalent) :

- ▶ générale récursive,
- ▶ calculable par machine de Turing,
- ▶ à voir → représentable dans le $\lambda$-calcul.

▶ Une partie $A \subseteq \mathbb{N}^k$ est dite **décidable** lorsque sa fonction indicatrice $\mathbb{N}^k \to \mathbb{N}$

$$\mathbf{1}_A\colon n \mapsto \begin{cases} 1 & \text{si } n \in A \\ 0 & \text{si } n \notin A \end{cases}$$

est calculable (répondre « oui » ou « non » selon que $n \in A$ ou $n \notin A$).

▶ Une partie $A \subseteq \mathbb{N}^k$ est dite **semi-décidable** lorsque sa fonction partielle « semi-indicatrice » $\mathbb{N} \dashrightarrow \mathbb{N}$ (d'ensemble de définition $A$)

$$n \mapsto \begin{cases} 1 & \text{si } n \in A \\ \uparrow & \text{si } n \notin A \end{cases}$$

est calculable (répondre « oui » ou « ... » selon que $n \in A$ ou $n \notin A$).

**Fluctuations terminologiques [transp. 62]**

▶ Synonymes de **calculable** pour une fonction partielle $\mathbb{N}^k \dashrightarrow \mathbb{N}$ :

- ▶ « semi-calculable » (réservant « calculable » pour les fonctions *totales*),
- ▶ « (générale) récursive ».

▶ Synonymes de **décidable** pour une partie $\subseteq \mathbb{N}^k$ :

- ▶ « calculable »,
- ▶ « récursive ».

▶ Synonymes de **semi-décidable** pour une partie $\subseteq \mathbb{N}^k$ :

- ▶ « semi-calculable »,
- ▶ « calculablement énumérable »,
- ▶ « récursivement énumérable ».

(La raison du mot « énumérable » sera expliquée après.)

**Décidable = semi-décidable de complémentaire semi-décidable [transp. 63]**

▶ Si $A \subseteq \mathbb{N}^k$ est décidable, alors son complémentaire $\complement A := \mathbb{N}^k \setminus A$ l'est aussi.
Preuve : échanger 0 et 1 dans la réponse. □

▶ Si $A$ est décidable, alors $A$ est semi-décidable.
Preuve : si réponse 0, faire une boucle infinie. □

▶ Donc : si $A$ est décidable, alors $A$ et $\complement A$ sont semi-décidables.

▶ La réciproque est également valable : si $A$ et $\complement A$ sont semi-décidables alors $A$ est décidable.

Idée : lancer « en parallèle » un algorithme qui semi-décide $A$ et un qui semi-décide $\complement A$ ; l'un des deux finira par donner la réponse voulue.

Mais que signifie « lancer en parallèle » ici ?

**Lancement en parallèle [transp. 64]**

On suppose que :

- $\varphi_{e_1}(\underline{x})\downarrow$ ssi $\underline{x} \in A$
- $\varphi_{e_2}(\underline{x})\downarrow$ ssi $\underline{x} \notin A$

Comment décider si $\underline{x} \in A$ en terminant à coup sûr ?

Grâce au *th. de la forme normale* (transp. 37) : il y a un prédicat $T$ p.r. tel que

- $\varphi_{e_1}(\underline{x})\downarrow$ ssi $\exists n \in \mathbb{N}.\ T(n, e_1, \langle\langle \underline{x} \rangle\rangle)$
- $\varphi_{e_2}(\underline{x})\downarrow$ ssi $\exists n \in \mathbb{N}.\ T(n, e_2, \langle\langle \underline{x} \rangle\rangle)$

On a alors $\exists n \in \mathbb{N}.\ (T(n, e_1, \langle\langle \underline{x} \rangle\rangle)$ ou $T(n, e_2, \langle\langle \underline{x} \rangle\rangle))$ à coup sûr.

Algorithme (terminant à coup sûr) :

- parcourir $n = 0, 1, 2, 3, 4, \ldots$ (boucle non bornée),
- pour chacun, tester si $T(n, e_1, \langle\langle \underline{x} \rangle\rangle)$ et si $T(n, e_2, \langle\langle \underline{x} \rangle\rangle)$,
- si le premier vaut, renvoyer « oui, $\underline{x} \in A$ », si le second vaut, renvoyer « non, $\underline{x} \notin A$ » (sinon, continuer la boucle).

**Lancement en parallèle (variante machines de Turing) [transp. 65]**

On suppose que :

- la machine $M_1$ s'arrête sur $\underline{x}$ ssi $\underline{x} \in A$
- la machine $M_2$ s'arrête sur $\underline{x}$ ssi $\underline{x} \notin A$

Comment décider si $\underline{x} \in A$ en s'arrêtant à coup sûr ?

On va simuler $M_1$ et $M_2$ pour $n$ étapes jusqu'à ce que l'une d'elles s'arrête.

Algorithme (terminant à coup sûr) :

- parcourir $n = 0, 1, 2, 3, 4, \ldots$ (boucle non bornée),
- pour chacun, tester si l'exécution de $M_1$ s'arrête sur $\underline{x}$ en $\leq n$ étapes et si l'exécution de $M_2$ s'arrête sur $\underline{x}$ en $\leq n$ étapes,
- si le premier vaut, renvoyer « oui, $\underline{x} \in A$ », si le second vaut, renvoyer « non, $\underline{x} \notin A$ » (sinon, continuer la boucle).

C'est *exactement la même chose* que dans le transp. précédent, avec un nombre d'étapes d'exécution $n$ au lieu d'un arbre de calcul (détail sans importance).

### Problème de l'arrêt [transp. 66]

Le **problème de l'arrêt** est :

$$\mathscr{H} := \{(e, x) \in \mathbb{N}^2 : \varphi_e(x) \downarrow\}$$

▶ Il *n'est pas décidable* (transp. 45).

▶ Il *est* semi-décidable (par universalité : donné $(e, x)$, on peut exécuter $\varphi_e(x)$, et, s'il termine, renvoyer « oui »).

▶ Donc $\complement\mathscr{H}$ n'est pas semi-décidable.

▶ Toutes sortes de variantes possibles, p.ex. :
- ▶ $\{e \in \mathbb{N} : \varphi_e(e) \downarrow\}$ n'est pas décidable (preuve dans transp. 46)
- ▶ $\{e \in \mathbb{N} : \varphi_e(0) \downarrow\}$ n'est pas décidable (théorème s-m-n : $\varphi_e(x) = \varphi_{s(e,x)}(0)$ avec $s$ p.r. ; cf. transp. 58)

### Image d'un ensemble décidable [transp. 67]

▶ Si $A \subseteq \mathbb{N}$ est décidable et $f : \mathbb{N} \to \mathbb{N}$ (totale) calculable, alors l'image

$$f(A) := \{f(i) : i \in A\}$$

est semi-décidable.

Preuve : donné $m \in \mathbb{N}$, pour semi-décider si $m \in f(A)$, parcourir $i = 0, 1, 2, 3 \ldots$, et pour chacun, décider si $i \in A$ et, si oui, calculer $f(i)$ et comparer à $m$. Si $i \in A$ et $f(i) = m$, renvoyer « oui » ; sinon, continuer la boucle. □

▶ Réciproquement, si $B \subseteq \mathbb{N}$ est semi-décidable, il existe $A \subseteq \mathbb{N}$ décidable et $f : \mathbb{N} \to \mathbb{N}$ (totale) calculable tels que $B = f(A)$.

Preuve : soit $e$ tel que $B = \{m : \varphi_e(m) \downarrow\}$ ; soit $A$ l'ensemble des $\langle n, m\rangle$ tels que $T(n, e, \langle\langle m\rangle\rangle)$ : alors $A$ est décidable et son image par $\langle n, m\rangle \mapsto m$ est $B$. □

Redite : soit $M$ une machine de Turing qui s'arrête sur $m$ ssi $m \in B$ ; soit $A$ l'ensemble des $\langle n, m\rangle$ tels que $M$ s'arrête sur $m$ en $\leq n$ étapes : alors $A$ est décidable et son image par $\langle n, m\rangle \mapsto m$ est $B$. □

▶ Variante : $B \subseteq \mathbb{N}$ *non vide* est semi-décidable ssi il existe $f : \mathbb{N} \to \mathbb{N}$ totale calculable telle que $f(\mathbb{N}) = B$. D'où le terme « calculablement énumérable ».

### Stabilités par opérations booléennes [transp. 68]

Les ensembles **décidables** sont stables par :

- ▶ réunions finies,
- ▶ intersections finies,
- ▶ complémentaire,
- ▶ *mais pas par* projection $\mathbb{N}^k \to \mathbb{N}^{k'}$ (où $k' \leq k$). (Le problème de l'arrêt est une projection d'un ensemble décidable, cf. transp. précédent.)

Les ensembles **semi-décidables** sont stables par :

- ▶ réunions finies (par lancement en parallèle !),
- ▶ intersections finies,
- ▶ projection $\mathbb{N}^k \to \mathbb{N}^{k'}$ (où $k' \leq k$), (Les ensembles semi-décidables sont projections d'ensembles décidables donc sont eux-mêmes stables par projections, cf. transp. précédent et idées proches.)
- ▶ *mais pas par complémentaire.*

### Le théorème de Rice : énoncé [transp. 69]

Soit $\mathbf{R}^{(1)}$ l'ensemble des fonctions partielles $\mathbb{N} \dashrightarrow \mathbb{N}$ calculables (= générales récursives), et $\Phi\colon e \mapsto \varphi_e^{(1)}$ qui définit une surjection $\mathbb{N} \to \mathbf{R}^{(1)}$.

Si $e$ est l'« intention » (l'algorithme, le programme), alors $\Phi(e)$ est l'« extension » (la fonction, i.e., son graphe) définie par $e$.

▶ **Théorème** (Rice) : si $F \subseteq \mathbf{R}^{(1)}$ est un ensemble de fonctions partielles tel que $\Phi^{-1}(F) := \{e \in \mathbb{N} : \varphi_e^{(1)} \in F\}$ est *décidable*, alors $F = \varnothing$ ou $F = \mathbf{R}^{(1)}$.

**Moralité :** *aucune propriété non-triviale* de la fonction $\varphi_e^{(1)}$ calculée par un programme *n'est décidable* en regardant le programme $e$.

Exemples :

- ▶ $\{e \in \mathbb{N} : \varphi_e^{(1)}(0)\downarrow\}$ n'est pas décidable ($\Rightarrow$ Rice *généralise* l'indécidabilité du pb. de l'arrêt).
- ▶ $\{e \in \mathbb{N} : \varphi_e^{(1)}$ totale$\}$ n'est pas décidable.
- ▶ $\{e \in \mathbb{N} : \forall n.\, (\varphi_e^{(1)}(n)\downarrow \Rightarrow \varphi_e^{(1)}(n) = 0)\}$ n'est pas décidable.

### Le théorème de Rice : preuve par théorème de récursion [transp. 70]

$\mathbf{R}^{(1)} = \{f\colon \mathbb{N} \dashrightarrow \mathbb{N} : f \text{ calculable}\}$

▶ **Théorème** (Rice) : si $F \subseteq \mathbf{R}^{(1)}$ est un ensemble de fonctions partielles tel que $\Phi^{-1}(F) := \{e \in \mathbb{N} : \varphi_e^{(1)} \in F\}$ est *décidable*, alors $F = \varnothing$ ou $F = \mathbf{R}^{(1)}$.

La preuve est très analogue à celle de l'indécidabilité du problème de l'arrêt.

Preuve : Supposons par l'absurde $\Phi^{-1}(F)$ décidable avec $F \neq \varnothing$ et $F \neq \mathbf{R}^{(1)}$. Soient $f \in F$ et $g \notin F$. Soit

$$h(e,x) := \begin{cases} f(x) & \text{si } e \notin \Phi^{-1}(F) \\ g(x) & \text{si } e \in \Phi^{-1}(F) \end{cases}$$

Alors $h\colon \mathbb{N}^2 \dashrightarrow \mathbb{N}$ est calculable par hypothèse (on peut décider si $e \in \Phi^{-1}(F)$). Par le théorème de récursion de Kleene (transp. 40), il existe $e$ tel que

$$\varphi_e^{(1)}(x) = h(e,x)$$

Si $e \in \Phi^{-1}(F)$ alors $h(e,x) = g(x)$ pour tout $x$, donc $\Phi(e) = g$ donc $e \notin \Phi^{-1}(F)$, une contradiction. Si $e \notin \Phi^{-1}(F)$ alors $h(e,x) = f(x)$ pour tout $x$, donc $\Phi(e) = f$ donc $e \in \Phi^{-1}(F)$, une contradiction. □

### Réductions : introduction [transp. 71]

▶ Situation typique : on veut montrer qu'une question $D$ (« problème de décision », souvent déjà semi-décidable) est indécidable. Ceci se fait typiquement en *réduisant le problème de l'arrêt* à $D$, c'est-à-dire :

« Supposons par l'absurde que $D$ soit décidable, c'est-à-dire que j'ai un algorithme qui répond à la question $D$ (comprendre : "$n \in D$ ?").

Je montre qu'*en utilisant cet algorithme* je peux résoudre le problème de l'arrêt.

Ceci est une contradiction (car le problème de l'arrêt est indécidable),
donc $D$ est indécidable. »

▶ Les notions de réduction formalisent cet argument : intuitivement,

« $A$ se réduit à $B$ »
signifie
« si $B$ est décidable alors $A$ est décidable »
(mais constructivement)

### Le théorème de Rice : preuve par réduction (1/2) [transp. 72]

$\mathbf{R}^{(1)} = \{f : \mathbb{N} \dashrightarrow \mathbb{N} : f \text{ calculable}\}$

▶ **Théorème** (Rice) : si $F \subseteq \mathbf{R}^{(1)}$ est tel que $F \neq \varnothing$ et $F \neq \mathbf{R}^{(1)}$, alors $\Phi^{-1}(F) := \{e \in \mathbb{N} : \varphi_e^{(1)} \in F\}$ *n'est pas décidable.*

Preuve : Soit $F \subseteq \mathbf{R}^{(1)}$ avec $F \neq \varnothing$ et $F \neq \mathbf{R}^{(1)}$. Quitte à remplacer $F$ par $\complement F$, o.p.s. $\uparrow \notin F$ où $\uparrow$ est la fonction nulle part définie. Soit $f \in F$ où $f = \varphi_a^{(1)}$.
Pour $(e, x) \in \mathbb{N}^2$, considérons l'algorithme suivant, prenant en entrée $m \in \mathbb{N}$ :

- ▶ simuler $\varphi_e^{(1)}(x)$ avec la machine universelle, puis, si l'exécution termine,
- ▶ calculer $f(m) = \varphi_a^{(1)}(m)$ et (si l'exécution termine) renvoyer sa valeur.

Soit $b(e, x)$ le code de l'algorithme qu'on vient de décrire :

$$\varphi_{b(e,x)}^{(1)} = f \text{ si } \varphi_e^{(1)}(x) \downarrow \quad \text{et} \quad \varphi_{b(e,x)}^{(1)} = \uparrow \text{ si } \varphi_e^{(1)}(x) \uparrow$$

notamment $\varphi_{b(e,x)}^{(1)} \in F$ ssi $\varphi_e^{(1)}(x) \downarrow$ (← c'est là la réduction). .../...

### Le théorème de Rice : preuve par réduction (2/2) [transp. 73]

$\mathbf{R}^{(1)} = \{f : \mathbb{N} \dashrightarrow \mathbb{N} : f \text{ calculable}\}$ ; on a supposé $F \subseteq \mathbf{R}^{(1)}$ avec $\uparrow \notin F$ et $f \in F$

On a construit (transp. précédent) un $b(e, x)$, avec $b : \mathbb{N}^2 \to \mathbb{N}$ calculable (même p.r.) tel que $\varphi_{b(e,x)}^{(1)} \in F$ ssi $\varphi_e^{(1)}(x) \downarrow$, c'est-à-dire

$$b(e, x) \in \Phi^{-1}(F) \iff (e, x) \in \mathscr{H}$$

où $\mathscr{H} := \{(e, x) \in \mathbb{N}^2 : \varphi_e(x) \downarrow\}$ est le problème de l'arrêt.

Si $\Phi^{-1}(F)$ était décidable, alors $\mathscr{H}$ le serait aussi, par l'algorithme :

- ▶ donnés $e, x$, calculer $b(e, x)$, décider si $b(e, x) \in \Phi^{-1}(F)$,
- ▶ si oui, répondre « oui », sinon répondre « non ».

Or $\mathscr{H}$ n'est pas décidable, donc $\Phi^{-1}(F)$ non plus. □

On dit qu'on a *réduit le problème de l'arrêt* à $\Phi^{-1}(F)$ (*via* la fonction $b$).

### Réduction « many-to-one » [transp. 74]

**Définition :** Si $A, B \subseteq \mathbb{N}$, on note $A \leq_{\mathrm{m}} B$ lorsqu'il existe $\rho : \mathbb{N} \to \mathbb{N}$ *calculable totale* telle que

$$\rho(m) \in B \iff m \in A$$

(c'est-à-dire $A = \rho^{-1}(B)$).

**Intuitivement :** si j'ai un gadget qui répond à la question "$n \in B$ ?", je peux répondre à la question "$m \in A$ ?" en transformant $m$ en $\rho(m) =: n$ et en utilisant le gadget (une seule fois, à la fin).

**Clairement**, si $A \leq_{\mathrm{m}} B$ avec $B$ décidable (resp. semi-décidable), alors $A$ est décidable (resp. semi-décidable).

*Notamment*, si $\mathscr{H} \leq_{\mathrm{m}} D$ alors $D$ *n'est pas* décidable.

La relation $\leq_{\mathrm{m}}$ est réflexive et transitive (c'est un « préordre ») ; la relation $\equiv_{\mathrm{m}}$ définie par $A \equiv_{\mathrm{m}} B$ ssi $A \leq_{\mathrm{m}} B$ et $B \leq_{\mathrm{m}} A$ est une relation d'équivalence, les classes pour laquelle s'appellent « degrés many-to-one » et sont partiellement ordonnés par $\leq_{\mathrm{m}}$.

### Réduction de Turing : présentation informelle [transp. 75]

**Informellement :** Si $A, B \subseteq \mathbb{N}$, on note $A \leq_{\mathrm{T}} B$ s'il existe un algorithme qui

- prend en entrée $m \in \mathbb{N}$,
- peut à tout moment demander à savoir si $n \in B$ (« interroger l'oracle »),
- termine en temps fini,
- et renvoie « oui » si $m \in A$, et « non » si $m \notin A$.

**Intuitivement :** à la différence de la réduction many-to-one où on ne peut poser la question "$n \in B$ ?" que sur une seule valeur $\rho(n)$ à la fin du calcul, ici on peut interroger l'oracle de façon libre et illimitée (mais finie !) au cours de l'algorithme.

La relation $A \leq_{\mathrm{T}} B$ est beaucoup plus faible que $A \leq_{\mathrm{m}} B$.

Par exemple, $(\complement B) \leq_{\mathrm{T}} B$ pour tout $B \subseteq \mathbb{N}$ (savoir décider "$n \in B$ ?" permet évidemment de décider "$n \notin B$ ?"), alors que $(\complement\mathscr{H}) \not\leq_{\mathrm{m}} \mathscr{H}$ car $\complement\mathscr{H}$ n'est pas semi-décidable.

Mais comment formaliser cette « interrogation » ?

### Réduction de Turing : formalisation(s) possible(s) [transp. 76]

Comment définir $A \leq_{\mathrm{T}} B$ pour $A, B \subseteq \mathbb{N}$ ? (I.e., « $A$ est calculable en utilisant $B$ ».)

**Formalisation 1 :** la fonction indicatrice $\mathbf{1}_A$ de $A$ appartient à la plus petite classe de fonctions qui contient les projections, les constantes, la fonction successeur *et la fonction indicatrice* $\mathbf{1}_B$ *de* $B$ et stable par composition, récursion primitive et opérateur $\mu$.

**Formalisation 2 :** il existe une fonction calculable qui prend en entrée $m \in \mathbb{N}$ et une liste $\langle\langle\langle n_0, \mathbf{1}_B(n_0)\rangle, \dots, \langle n_k, \mathbf{1}_B(n_k)\rangle\rangle\rangle$ de réponses à des questions "$n \in B$ ?", et qui (si ces réponses cont correctes !) termine et renvoie

- soit une réponse finale à la question "$m \in A$ ?" (disons encodée comme $\langle 0, \mathbf{1}_A(m)\rangle$),
- soit une nouvelle interrogation "$n \in B$ ?" (disons encodée comme $\langle 1, n\rangle$),

de sorte que si on commence par $k = 0$ et qu'on ajoute à chaque fois la réponse correcte $\langle n_{k+1}, \mathbf{1}_B(n_{k+1})\rangle$ à l'interrogation $\langle 1, n_{k+1}\rangle$ posée, alors la fonction finit par produite la réponse finale correcte $(\langle 0, \mathbf{1}_A(m)\rangle)$.

**Réduction de Turing : quelques propriétés [transp. 77]**

**Clairement**, si $A \leq_{\mathrm{T}} B$ avec $B$ décidable, alors $A$ est décidable.
(Ceci *ne vaut pas* pour « semi-décidable ».)

*Notamment*, si $\mathscr{H} \leq_{\mathrm{T}} D$ alors $D$ *n'est pas* décidable.

La relation $\leq_{\mathrm{T}}$ est réflexive et transitive (c'est un « préordre ») ; la relation $\equiv_{\mathrm{T}}$ définie par $A \equiv_{\mathrm{T}} B$ ssi $A \leq_{\mathrm{T}} B$ et $B \leq_{\mathrm{T}} A$ est une relation d'équivalence, les classes pour laquelle s'appellent « degrés de Turing » et sont partiellement ordonnés par $\leq_{\mathrm{T}}$.

Comme $A \leq_{\mathrm{m}} B$ implique $A \leq_{\mathrm{T}} B$, chaque degré de Turing est une réunion de degrés many-to-one (la relation d'équivalence $\equiv_{\mathrm{T}}$ est plus grossière que $\equiv_{\mathrm{m}}$).

Les parties décidables de $\mathbb{N}$ forment le plus petit degré de Turing, souvent noté $\mathbf{0}$. Le degré de Turing de $\mathscr{H}$ est noté $\mathbf{0}'$. (Il existe des ensembles de degré strictement compris entre $\mathbf{0}$ et $\mathbf{0}'$, même des ensembles semi-décidables, mais il semble qu'aucun n'apparaît naturellement.)

# 6 Le $\lambda$-calcul non typé

**Le $\lambda$-calcul : aperçu [transp. 78]**

Le **$\lambda$-calcul non typé** manipule des expressions du type

$$\lambda x.\lambda y.\lambda z.((xz)(yz))$$
$$\lambda f.\lambda x.f(f(f(f(f(fx)))))$$
$$(\lambda x.(xx))(\lambda x.(xx))$$

Ces expressions s'appelleront des **termes** du $\lambda$-calcul.

Il faut comprendre intuitivement qu'un terme représente une sorte de fonction qui prend une autre telle fonction en entrée et renvoie une autre telle fonction.

Deux constructions fondamentales :

- **application** : $(PQ)$ : appliquer la fonction $P$ à la fonction $Q$ ;
- **abstraction** : $\lambda v.E$ : créer la fonction qui prend un argument et le remplace pour $v$ dans l'expression $E$ (en gros $v \mapsto E$).

**Le $\lambda$-calcul : termes [transp. 79]**

- Un **terme** du $\lambda$-calcul est (inductivement) :
  - une **variable** ($a$, $b$, $c$... en nombre illimité),
  - une **application** $(PQ)$ où $P$ et $Q$ sont deux termes,
  - une **abstraction** $\lambda v.E$ où $v$ est une variable et $E$ un terme ; on dira que la variable $v$ est **liée** dans $E$ par ce $\lambda$.
- Conventions d'écriture :
  - l'application *n'est pas associative* : on parenthèse implicitement vers la gauche : « $xyz$ » signifie « $((xy)z)$ » ;
  - abréviation de plusieurs $\lambda$ : on note « $\lambda uv.E$ » pour « $\lambda u.\lambda v.E$ » ;
  - l'abstraction est moins prioritaire que l'application : « $\lambda x.xy$ » signifie $\lambda x.(xy)$ *pas* $(\lambda x.x)y$.
- Une variable non liée est dite **libre** : $(\lambda x.x)x$ (le dernier $x$ est libre).
- Un terme sans variable libre est dit **clos**.
- Les variables liées sont muettes : $\lambda x.x \equiv \lambda y.y$, comprendre $\lambda\bullet.\bullet$.

### Le $\lambda$-calcul : variables liées [transp. 80]

On appelle **$\alpha$-conversion** le renommage des variables liées : ces termes sont considérés comme équivalents.

- $\lambda x.x \equiv \lambda y.y$ et $\lambda xyz.((xz)(yz)) \equiv \lambda uvw.((uw)(vw))$
- Attention à *ne pas capturer* de variable libre : $\lambda y.xy \not\equiv \lambda x.xx$.
- En cas de synonymie, la variable est liée par le $\lambda$ le *plus intérieur* pour ce nom ($\cong$ portée lexicale) : $\lambda x.\lambda x.x \equiv \lambda x.\lambda v.v \not\equiv \lambda u.\lambda x.u$.
- Mieux vaut ne pas penser aux termes typographiquement, mais à chaque variable liée comme un *pointeur vers la $\lambda$-abstraction qui la lie* :
$$\lambda x.(\lambda y.y(\lambda z.z))(\lambda z.xz) \equiv \lambda\bullet.(\lambda\bullet.\bullet(\lambda\bullet.\bullet))(\lambda\bullet.\bullet\bullet)$$
- Autre convention possible : **indices de De Bruijn** : remplacer les variables liées par le numéro du $\lambda$ qui la lie, en comptant du plus intérieur (1) vers le plus extérieur :
$$\lambda x.(\lambda y.y(\lambda z.z))(\lambda z.xz) \equiv \lambda.(\lambda.1(\lambda.1))(\lambda.21)$$
deux termes sont $\alpha$-équivalents ssi leur écriture avec indice de De Bruijn est identique.

### Le $\lambda$-calcul : $\beta$-réduction [transp. 81]

On travaille désormais sur des termes à $\alpha$-équivalence près.

- Un **redex** (« reducible expression ») est un terme de la forme $(\lambda v.E)T$. Son **réduit** est le terme $E[v\backslash T]$ obtenu par remplacement de $T$ pour $v$ dans $E$, en évitant tout conflit de variables.

Exemples :

- $(\lambda x.xx)y \to yy$
- $(\lambda x.xx)(\lambda x.xx) \to (\lambda x.xx)(\lambda x.xx)$ (est son propre réduit)
- $(\lambda xy.x)z \to \lambda y.z$ (car $\lambda xy.x$ abrège $\lambda x.\lambda y.x$)
- $(\lambda xy.x)y \to \lambda y_1.y$ (attention au conflit de variable !)
- $(\lambda x.\lambda x.x)y \to \lambda x.x$ (car $\lambda x.\lambda x.x \equiv \lambda z.\lambda x.x$ : le $\lambda$ extérieur ne lie rien)

- Un terme n'ayant *pas de redex en sous-expression* est dit en **forme ($\beta$-)normale**. Ex. : $\lambda xyz.((xz)(yz))$.
- On appelle **$\beta$-réduction** le remplacement en sous-expression d'un redex par son réduit. Ex. : $\lambda x.(\lambda y.y(\lambda z.z))(\lambda z.xz) \to \lambda x.(\lambda z.xz)(\lambda z.z)$.

### Le $\lambda$-calcul : normalisation par $\beta$-réductions [transp. 82]

On note :

- $T \to T'$ (ou $T \to_\beta T'$) si $T'$ s'obtient par $\beta$-réduction d'un redex de $T$.
- $T \twoheadrightarrow T'$ (ou $T \twoheadrightarrow_\beta T'$) si $T'$ s'obtient par une suite finie de $\beta$-réductions ($T = T_0 \to \cdots \to T_n = T'$, y compris $n = 0$ soit $T' = T$).
- $T$ est **faiblement normalisable** lorsque $T \twoheadrightarrow T'$ avec $T'$ en forme normale (*une certaine* suite de $\beta$-réductions termine).

- $T$ est **fortement normalisable** lorsque *toute* suite de $\beta$-réductions termine (sur un terme en forme normale).

Exemples :

- $(\lambda x.xx)(\lambda x.xx)$ n'est pas faiblement normalisable (la $\beta$-réduction boucle).
- $(\lambda uz.z)((\lambda x.xx)(\lambda x.xx))$ n'est pas fortement normalisable mais il est faiblement normalisable $\to \lambda z.z$.
- $(\lambda uz.u)((\lambda t.t)(\lambda x.xx))$ est fortement normalisable $\twoheadrightarrow \lambda zx.xx$.

### Le $\lambda$-calcul : confluence et choix d'un redex [transp. 83]

▶ **Théorème** (Church-Rosser) : si $T \twoheadrightarrow T'_1$ et $T \twoheadrightarrow T'_2$ alors il existe $T''$ tel que $T'_1 \twoheadrightarrow T''$ et $T''_2 \twoheadrightarrow T''$.
En particulier, si $T'_1, T'_2$ sont en forme normale, alors $T'_1 \equiv T'_2$ (unicité de la normalisation).

Pour *éviter* ce théorème, on va faire un choix simple de redex à réduire :
▶ On appelle **redex extérieur gauche** d'un $\lambda$-terme le redex dont le $\lambda$ est *le plus à gauche*. Exemples : $\lambda x.x((\lambda y.y)x)$ ; $\lambda x.(\lambda y.(\lambda z.z)y)x$.

▶ On écrira $T \to_{\mathsf{lft}} T'$ lorsque $T'$ s'obtient par $\beta$-réduction du redex extérieur gauche, et $T \twoheadrightarrow_{\mathsf{lft}} T'$ pour une suite de telles réductions.

On peut montrer (mais on évitera d'utiliser) :
▶ **Théorème** (Curry &al) : si $T \twoheadrightarrow T'$ avec $T'$ en forme normale, alors $T \twoheadrightarrow_{\mathsf{lft}} T'$ (i.e., la réduction ext. gauche normalise les termes faiblement normalisables).

### Réduction extérieure gauche : exemples [transp. 84]

Divers noms utilisés : « réduction en ordre normal », « réduction gauche », etc.

On a noté $T \twoheadrightarrow_{\mathsf{lft}} T'$ lorsque $T'$ s'obtient par une succession de $\beta$-réductions à chaque fois du redex dont le $\lambda$ est le plus à gauche.

Exemples :

- $(\lambda x.xx)(\lambda x.xx) \to_{\mathsf{lft}} (\lambda x.xx)(\lambda x.xx) \to_{\mathsf{lft}} \cdots$ (boucle)
- $(\lambda uz.z)((\lambda x.xx)(\lambda x.xx)) = (\lambda u.\lambda z.z)((\lambda x.xx)(\lambda x.xx)) \to_{\mathsf{lft}} \lambda z.z$
- $(\lambda uz.u)((\lambda t.t)(\lambda x.xx)) = (\lambda u.\lambda z.u)((\lambda t.t)(\lambda x.xx)) \to_{\mathsf{lft}} \lambda z.((\lambda t.t)(\lambda x.xx)) \to_{\mathsf{lft}} \lambda z.\lambda x.xx = \lambda zx.xx$

Intérêt :

- cette stratégie de réduction est *déterministe*,
- (Curry &al :) si (« terme faiblement normalisant ») une réduction quelconque termine sur une forme normale, alors $\twoheadrightarrow_{\mathsf{lft}}$ le fait.

### Simulation du $\lambda$-calcul par les fonctions récursives [transp. 85]

▶ On peut coder un terme du $\lambda$-calcul sous forme d'entiers naturels.

▶ La fonction $T \mapsto 1$ qui à un terme $T$ associe 0 si $T$ est en forme normale et 1 si non, **est p.r.**

▶ La fonction $T \mapsto T'$ qui à un terme $T$ associe sa réduction extérieure gauche **est p.r.**

▶ Conséquence : la fonction $(n, T) \mapsto T^{(n)}$ qui à $n \in \mathbb{N}$ et un terme $T$ associe le terme obtenu après $n$ réductions extérieures gauches **est p.r.**

▶ La fonction qui à $T$ associe la forme normale (et/ou le nombre d'étapes d'exécution) *si la réduction extérieure gauche termine*, et $\uparrow$ (non définie) si elle ne termine pas, est **générale récursive**.

**Moralité :** les fonctions récursives peuvent simuler la réduction extérieure gauche du $\lambda$-calcul (ou n'importe quelle autre réduction, mais on se concentre sur celle-ci).

### Entiers de Church [transp. 86]

On définit les termes en forme normale $\overline{n} := \lambda fx.f^{\circ n}(x)$ pour $n \in \mathbb{N}$, c-à-d :

- ▶ $\overline{0} := \lambda fx.x$
- ▶ $\overline{1} := \lambda fx.fx$
- ▶ $\overline{2} := \lambda fx.f(fx)$
- ▶ $\overline{3} := \lambda fx.f(f(fx))$, etc.

Intuitivement, $\overline{n}$ prend une fonction $f$ et renvoie sa $n$-ième itérée.

▶ Posons $A := \lambda mfx.f(mfx) = \lambda m.\lambda f.\lambda x.f(mfx)$
Alors

$$\begin{aligned} A\overline{n} &= (\lambda m.\lambda f.\lambda x.f(mfx))(\lambda g.\lambda y.g^{\circ n}(y)) \\ &\to_{\mathsf{lft}} \lambda f.\lambda x.f(((\lambda g.\lambda y.g^{\circ n}(y)))fx) \\ &\to_{\mathsf{lft}}\to_{\mathsf{lft}} \lambda f.\lambda x.f(f^{\circ n}(x)) = \lambda f.\lambda x.f^{\circ(n+1)}(x) = \overline{n+1} \end{aligned}$$

### Calculs dans le $\lambda$-calcul : une convention [transp. 87]

On dira qu'une fonction $f\colon \mathbb{N}^k \dashrightarrow \mathbb{N}$ est **représentable par un $\lambda$-terme** lorsqu'il existe un terme clos $t$ tel que, pour tous $x_1, \dots, x_k \in \mathbb{N}$ :

- ▶ si $f(x_1, \dots, x_k)\downarrow = y$ alors $t\overline{x_1}\cdots\overline{x_k} \twoheadrightarrow_{\mathsf{lft}} \overline{y}$,
- ▶ si $f(x_1, \dots, x_k)\uparrow$ alors $t\overline{x_1}\cdots\overline{x_k} \to_{\mathsf{lft}} \cdots$ ne termine pas,

où $\overline{z}$ désigne l'entier de Church associé à $z \in \mathbb{N}$.

Exemples :

- ▶ $\lambda mfx.f(mfx)$ représente $m \mapsto m+1$ (transp. précédent),
- ▶ $\lambda mnfx.nf(mfx)$ représente $(m, n) \mapsto m+n$,
- ▶ $\lambda mnf.n(mf)$ représente $(m, n) \mapsto mn$ (itérer $n$ fois l'itérée $m$-ième),
- ▶ $\lambda mn.nm$ représente $(m, n) \mapsto m^n$ (itérer $n$ fois l'itération $m$-ième).
- ▶ $\lambda mnp.p(\lambda y.n)m$ représente $(m, n, p) \mapsto \begin{cases} m & \text{si } p = 0 \\ n & \text{si } p \geq 1 \end{cases}$ (itérer $p$ fois « remplacer par $n$ »).

### Représentation des fonctions p.r. : cas faciles [transp. 88]

(Cf. transp. 16.)
Fonction p.r. facilement représentables par un $\lambda$-terme :

- $\lambda x_1 \cdots x_k.x_i$ représente $(x_1, \ldots, x_k) \mapsto x_i$ ;
- $\lambda x_1 \cdots x_k.\overline{c}$ représente $(x_1, \ldots, x_k) \mapsto c$ ;
- $A := \lambda mfx.f(mfx)$ représente $x \mapsto x + 1$ ;
- si $v_1, \ldots, v_\ell$ représentent $g_1, \ldots, g_\ell$ et $w$ représente $h$, alors $\lambda x_1 \cdots x_k.w(v_1 x_1 \cdots x_k) \cdots (v_\ell x_1 \cdots x_k)$ représente $(x_1, \ldots, x_k) \mapsto h(g_1(x_1, \ldots, x_k), \ldots, g_\ell(x_1, \ldots, x_k))$ ;
- si $v$ représente $g$ et $w$ représente $h$, alors
$$\lambda x_1 \cdots x_k z.z(wx_1 \cdots x_k)(vx_1 \cdots x_k)$$
représente $f$ définie par la récursion primitive
$$f(x_1, \ldots, x_k, 0) = g(x_1, \ldots, x_k)$$
$$f(x_1, \ldots, x_k, z+1) = h(x_1, \ldots, x_k, f(x_1, \ldots, x_k, z))$$
*mais* on veut $f(x_1, \ldots, x_k, z+1) = h(x_1, \ldots, x_k, f(x_1, \ldots, x_k, z), z)$...?

### Représentation des couples d'entiers [transp. 89]

(Oublions $x_1, \ldots, x_k$ pour ne pas alourdir les notations.)
Comment passer de
$$\begin{cases} f(0) = g \\ f(z+1) = h(f(z)) \end{cases} \quad \text{à} \quad \begin{cases} f(0) = g \\ f(z+1) = h(f(z), z) \end{cases} \quad ?$$
On voudrait définir
$$\tilde{f}(z) = (f(z), z) \quad \text{soit} \quad \begin{cases} \tilde{f}(0) = (g, 0) \\ \tilde{f}(z+1) = \tilde{h}(\tilde{f}(z)) \end{cases} \quad \text{où } \tilde{h}(y, z) = (h(y, z), z+1)$$

On va définir (temporairement ?)
$$\overline{m,n} := \lambda fgx.f^{\circ m}(g^{\circ n}(x)) \quad \text{si } m, n \in \mathbb{N}$$
$$\Pi := \lambda mnfgx.(mf)(ngx) \quad \text{donc } \Pi\overline{m}\,\overline{n} \twoheadrightarrow_{\mathsf{lft}} \overline{m,n}$$
$$\pi_1 := \lambda pfx.pf(\lambda z.z)x \quad \text{donc } \pi_1\overline{m,n} \twoheadrightarrow_{\mathsf{lft}} \overline{m}$$
$$\pi_2 := \lambda pgx.p(\lambda z.z)gx \quad \text{donc } \pi_2\overline{m,n} \twoheadrightarrow_{\mathsf{lft}} \overline{n}$$

### Représentation de la récursion primitive générale [transp. 90]

Maintenant qu'on a une représentation des couples d'entiers naturels dans le $\lambda$-calcul donnée par $\Pi$ (formation de paires) et $\pi_1, \pi_2$ (projections).

- Si $v$ représente $g\colon \mathbb{N}^k \dashrightarrow \mathbb{N}$ et $w$ représente $h\colon \mathbb{N}^{k+2} \dashrightarrow \mathbb{N}$, alors $f\colon \mathbb{N}^{k+1} \dashrightarrow \mathbb{N}$ est représentée par

$$\lambda x_1 \cdots x_k z.\pi_1(z(\lambda p.\Pi(wx_1 \cdots x_k(\pi_1 p)(\pi_2 p))A(\pi_2 p))(\Pi(vx_1 \cdots x_k)\overline{0}))$$

où
$$f(x_1, \ldots, x_k, 0) = g(x_1, \ldots, x_k)$$
$$f(x_1, \ldots, x_k, z+1) = h(x_1, \ldots, x_k, f(x_1, \ldots, x_k, z), z)$$
(toujours avec $A := \lambda mfx.f(mfx)$).

D'autres encodages des paires sont possibles et possiblement plus simples, p.ex., $\Pi := \lambda rsa.ars$ et $\pi_1 := \lambda p.p(\lambda rs.r)$ et $\pi_2 := \lambda p.p(\lambda rs.s)$ (fonctionnent sur plus que les entiers de Church).

Bref, (au moins) *les fonctions p.r. sont représentables par $\lambda$-termes.*

**Le combinateur $\mathsf{Y}$ de Curry [transp. 91]**

▶ Pour représenter toutes les fonctions récursives, on va implémenter les appels récursifs dans le $\lambda$-calcul.

▶ Pour ça, on va utiliser la même idée que le théorème de récursion de Kleene (transp. 25).

Posons

$$\mathsf{Y} := \lambda f.((\lambda x.f(xx))(\lambda x.f(xx)))$$

Idée :

$$\begin{aligned}\mathsf{Y} &:= \lambda f.((\lambda x.f(xx))(\lambda x.f(xx)))\\ &\to \lambda f.f((\lambda x.f(xx))(\lambda x.f(xx)))\\ &\to \lambda f.f(f((\lambda x.f(xx))(\lambda x.f(xx)))) \to \cdots\end{aligned}$$

▶ Le terme (non normalisable !) $\mathsf{Y}$ "**recherche**" un point fixe de son argument.

▶ Permet d'implémenter les appels récursifs (transp. suivant).

**Récursion avec le combinateur $\mathsf{Y}$ [transp. 92]**

▶ On veut implémenter une définition par appels récursifs dans le $\lambda$-calcul, `let rec` $h = (\ldots h \ldots)$, disons `let rec` $h = E$ où $E = (\ldots h \ldots)$ est un terme faisant intervenir $h$.

L'idée est comme dans le transp. 42.

▶ On considère le terme :

$$\begin{aligned}\mathsf{Y}(\lambda h.E) &= (\lambda f.((\lambda x.f(xx))(\lambda x.f(xx))))(\lambda h.E)\\ &\to (\lambda x.(\lambda h.E)(xx))(\lambda x.(\lambda h.E)(xx)) =: h_{\text{fix}}\\ &\to (\lambda h.E)((\lambda x.(\lambda h.E)(xx))(\lambda x.(\lambda h.E)(xx))) = (\lambda h.E)(h_{\text{fix}})\\ &\to E[h\backslash h_{\text{fix}}] \text{ (substitution de } h_{\text{fix}} \text{ pour } h \text{ dans } E)\end{aligned}$$

Donc $h_{\text{fix}}$ (et donc $\mathsf{Y}(\lambda h.E)$) se comporte, à des $\beta$-réductions près, comme la fonction récursive recherchée.

▶ Si l'évaluation (i.e., la $\beta$-réduction) de $E$ termine et ne fait pas intervenir $h$, alors $h_{\text{fix}}$ donne juste $E$, sinon elle itère avec $h_{\text{fix}}$ pour $h$ jusqu'à ce que ce soit le cas : c'est bien ce que fait un appel récursif.

**Digression / variante : le combinateur $\mathsf{Z}$ [transp. 93]**

▶ Le bon fonctionnement du combinateur $\mathsf{Y}$ dépend du fait que la stratégie de $\beta$-réduction utilisée est extérieure gauche. Sinon le redex $(\lambda x.f(xx))(\lambda x.f(xx))$ peut causer un cycle de $\beta$-réductions.

▶ La variante suivante évite ce problème pour définir une fonction par appels récursifs dans un langage qui n'évalue pas « dans les $\lambda$ » :

$$\mathsf{Z} := \lambda f.((\lambda x.f(\lambda v.xxv))(\lambda x.f(\lambda v.xxv)))$$

Cette fois :

$$\begin{aligned}\mathsf{Z}(\lambda h.E) &= (\lambda f.((\lambda x.f(\lambda v.xxv))(\lambda x.f(\lambda v.xxv))))(\lambda h.E)\\ &\to (\lambda x.(\lambda h.E)(\lambda v.xxv))(\lambda x.(\lambda h.E)(\lambda v.xxv)) =: h_{\text{fix}}\\ &\to (\lambda h.E)(\lambda v.(\lambda x.(\lambda h.E)(xx))(\lambda x.(\lambda h.E)(xx))v) = (\lambda h.E)(\lambda v.h_{\text{fix}}v)\\ &\to E[h\backslash \lambda v.h_{\text{fix}}v] \text{ (substitution de } \lambda v.h_{\text{fix}}v \text{ pour } h \text{ dans } E)\end{aligned}$$

La forme $\lambda v.h_{\text{fix}}v$ maintient $h_{\text{fix}}$ non-évalué (exemple transp. suivant).

### Digression (suite) : exemple en Scheme [transp. 94]

On prend ici l'exemple de Scheme pour avoir affaire à un langage fonctionnel non typé (le typage empêche l'implémentation directe du combinateur Y ou Z en OCaml ou Haskell).

▶ Récursion sans combinateurs :

```
(define proto-fibonacci   (lambda (self) ; Pass me as argument!     (lambda
(n)       (if (<= n 1) n           (+ ((self self) (- n 1)) ((self
self) (- n 2)))))))) (define fibonacci (proto-fibonacci proto-fibonacci))
(map fibonacci '(0 1 2 3 4 5 6)) → (0 1 1 2 3 5 8)
```

▶ L'idée est ici exactement celle de l'astuce de Quine : pour m'appeler « moi-même », je m'attends à me recevoir moi-même en argument, et je reproduis ceci lors de l'appel.

▶ Le combinateur Y (ou Z) automatise cette construction.

### Digression (suite) : exemple en Scheme [transp. 95]

```
;; (define y-combinator ;;   (lambda (f) ;;     ((lambda (x) (f (x
x))) (lambda (x) (f (x x)))))) (define z-combinator   (lambda (f)     ((lambda
(x) (f (lambda (v) ((x x) v))))      (lambda (x) (f (lambda (v) ((x
x) v)))))))) (define pre-fibonacci   (lambda (fib)     (lambda (n)       (if
(<= n 1) n           (+ (fib (- n 1)) (fib (- n 2)))))))) (define fibonacci
(z-combinator pre-fibonacci))
```

### Représentation de l'opérateur $\mu$ de Kleene [transp. 96]

Rappel : $\mu g(x_1, \ldots, x_k)$ est le plus petit $z$ tel que $g(z, x_1, \ldots, x_k) = 0$ et $g(i, x_1, \ldots, x_k) \downarrow$ pour $0 \leq i < z$, s'il existe.

▶ On peut faire l'algorithme « rechercher à partir de $z$ » par appels récursifs :

$$h(z, x_1, \ldots, x_k) = \begin{cases} z & \text{si } g(z, x_1, \ldots, x_k) = 0 \\ h(z+1, x_1, \ldots, x_k) & \text{(récursivement) sinon} \end{cases}$$

Alors $\mu g(x_1, \ldots, x_k) = h(0, x_1, \ldots, x_k)$.

▶ $T := \lambda pmn.p(\lambda y.n)m$ représente $(p, m, n) \mapsto \begin{cases} m & \text{si } p = 0 \\ n & \text{si } p \geq 1 \end{cases}$

▶ $A := \lambda mfx.f(mfx)$ représente $z \mapsto z + 1$

▶ La récursion est implémentée avec le combinateur Y :

$$\begin{aligned} &\mathsf{Y}(\lambda hzx_1 \cdots x_k.T(vzx_1 \cdots x_k)z(h(Az)x_1 \cdots x_k)) \\ \to\ &\mathsf{Y}(\lambda hzx_1 \cdots x_k.(vzx_1 \cdots x_k)(\lambda y.h(Az)x_1 \cdots x_k)z) \end{aligned}$$

### Équivalence entre $\lambda$-calcul et fonctions récursives [transp. 97]

▶ Toute fonction générale récursive (i.e., *calculable* !) $\mathbb{N}^k \dashrightarrow \mathbb{N}$ est représentée par un terme du $\lambda$-calcul (sous les conventions données : application aux entiers de Church, réduction extérieure gauche).

▶ Réciproquement, toute fonction $\mathbb{N}^k \dashrightarrow \mathbb{N}$ représentable par un terme du $\lambda$-calcul est calculable, car on peut implémenter la réduction extérieure gauche.

▶ Bref, $f\colon \mathbb{N}^k \dashrightarrow \mathbb{N}$ est représentable par un terme du $\lambda$-calcul *ssi* elle est générale récursive.

▶ De plus, cette équivalence est *constructive* : il existe des fonctions p.r. :

- ▶ l'une prend en entrée le numéro $e$ d'une fonction générale récursive (et l'arité $k$) et renvoie le code d'un terme du $\lambda$-calcul qui représente cette $\varphi_e^{(k)}$,
- ▶ l'autre prend en entrée le code d'un terme du $\lambda$-calcul qui représente une fonction $f$, et son arité $k$, et renvoie un numéro $e$ de $f$ dans les fonctions générales récursives $f = \varphi_e^{(k)}$.

# 7 Conclusion

**Récapitulation [transp. 98]**

▶ **Théorème** : les fonctions $\mathbb{N}^k \dashrightarrow \mathbb{N}$ **(1)** générales récursives, **(2)** représentables en $\lambda$-calcul, et **(3)** calculables par machine de Turing, coïncident toutes.

On les appelle les **fonctions calculables**.

▶ De plus, ces équivalences sont constructives : on peut passer algorithmiquement (= calculablement !) d'une représentation à l'autre.
Ce sont des formes de *compilation* d'un langage en un autre.

▶ Les questions suivantes *ne sont pas décidables* algorithmiquement :

- ▶ savoir si une machine de Turing donnée s'arrête,
- ▶ savoir si un terme du $\lambda$-calcul est normalisable.

**Turing-complétude [transp. 99]**

Un langage de programmation est dit **Turing-complet** lorsque (convenablement idéalisé !) il permet d'implémenter précisément les fonctions calculables au sens de Church-Turing.

Un ordinateur réel ne peut *jamais faire plus* qu'une machine de Turing (sauf p.-ê. : faire du hasard vrai). La question est de savoir si le langage permet *autant*.

Tous les langages de programmation généralistes *sont Turing-complets* : Python, Java, JavaScript, C, C++, OCaml, Haskell, Lisp, Perl, Ruby, Smalltalk, Prolog...

Certains le sont même plus ou moins « par accident » : CSS, TeX, XSLT, m4... (parfois sous conditions, ou sous réserve d'interprétation).

Pas toujours clair : assembleurs (pas évident d'idéaliser les entiers).

Conséquence du problème de l'arrêt : on ne peut pas algorithmiquement décider si un programme donné (en Python, etc.) termine ou non.

### Turing tarpit [transp. 100]

« Fosse à bitume de Turing » ?

▶ Tous les langages usuels se valent du point de vue de la calculabilité. Ce n'est pas pour autant qu'ils se valent en pratique ! (En commodité et/ou efficacité.)

▶ Ça ne signifie pas qu'un langage Turing-complet peut forcément « tout » faire. Par exemple, un langage qui ne permet comme entrée/sortie que d'afficher des entiers peut être Turing-complet et ne permet pas d'écrire « bonjour ».

▶ Si en principe on peut convertir toute fonction calculable dans tout langage Turing-complet, la conversion peut devenir extrêmement inefficace, malcommode ou illisible.

### $\lambda$-calcul non typé et type récursif [transp. 101]

Remarque faite pour plus tard.

Le fait d'avoir un type $t$ tel que $t \cong (t \to t)$ permet d'implémenter dans ce type le $\lambda$-calcul « non typé » (donc tue l'espoir de décider la terminaison).

Exemple en OCaml (ici, `loop` produit une boucle infinie) :

```
type t = T of (t -> t) let apply : t -> t -> t = fun (T rator) -> fun rand -> rator
rand let id : t = T (fun x -> x)  (* λx.x *) let ch0 : t = T (fun f -> T (fun x ->
x)) (* λfx.x *) let ch1 : t = T (fun f -> T (fun x -> apply f x)) (* λfx.fx *) let
ch2 : t = T (fun f -> T (fun x -> apply f (apply f x))) (* λfx.f(fx) *) let om : t
= T (fun x -> apply x x)(* λx.xx *) let loop : t = apply om om(* (λx.xx)(λx.xx) *)
(* let loop = (fun (T h) -> h (T h)) (T (fun (T h) -> h (T h))) *)
```

Remarquer qu'ici on arrive à provoquer une boucle infinie sans aucun `let rec` (et malgré le typage).

### Une méditation googologique [transp. 102]

Ceci est une sorte de digression, pour inviter à la réflexion.

« googologie » = étude des grands nombres ; de « googol », nom fantaisiste de $10^{100}$

On cherche à minorer calculabl$^{t}$ la fonction « castor affairé », c-à-d :

- ▶ concevoir un programme dans un langage de programmation idéalisé (machine de Turing, $\lambda$-calcul, Python, OCaml...),
- ▶ de taille « humainement raisonnable » (peu importent les détails),
- ▶ qui *termine en temps fini* (théoriquement !),
- ▶ mais calcule un nombre aussi grand que possible (variante : attend un temps aussi long que possible).

Exemple : implémenter $A_\Delta : n \mapsto A(n,n,n)$ (fonction d'Ackermann diagonale) et calculer $A_\Delta(A_\Delta(\cdots(100))) = A_\Delta^{\circ 100}(100)$ ou qqch du genre.
...On peut faire beaucoup plus grand !