Automates et Calculabilité

– Autres formalismes de calculablité –

Leçon 14 : 2 fév 2004

Uwe Nestmann

EPFL-I&C

Ackermann (I)

La fonction A (Ackermann, 1928) est totale et calculable.

$$\begin{aligned} A(0,y) &\stackrel{\text{def}}{=} y{+}1 \\ A(x{+}1,0) &\stackrel{\text{def}}{=} A(x,1) \\ A(x{+}1,y{+}1) &\stackrel{\text{def}}{=} A(x,A(x{+}1,y)) \end{aligned}$$

Pour tout x, la fonction A_x est primitive récursive.

$$\begin{aligned} A_0(y) &\stackrel{\text{def}}{=} y{+}1 \\ A_{x+1}(0) &\stackrel{\text{def}}{=} A_x(1) \\ A_{x+1}(y{+}1) &\stackrel{\text{def}}{=} A_x(A(x{+}1,y)) \end{aligned}$$

Toute fonction primitive récursive est « dominée » par une A_x.
La fonction A n'est dominée par aucune fonction A_x.
Par conséquent, A ne peut ***pas*** être ***primitive récursive***.
Intuitivement cela est du à la récursion sur 2(!) variables.

Ackermann (II)

Lemme:
Si $f : \mathbb{N} \to \mathbb{N}$ est une fonction primitive-récursive, alors
il existe $m, k \in \mathbb{N}$ tels que pour tout $n \geq k$: $A(m,n) > f(n)$.

Lemme:
Pour tout $m, n \in \mathbb{N}$: $A(m{+}1,n) > A(m,n)$.

Exercice:

- ☐ Définir $A'(n) \stackrel{\text{def}}{=} A(n,n)$.
- ☐ Suivre l'argumentation suivante:
  - Si A était primitive-récursive, alors A' le serait aussi.
  - Appliquer le premier lemme ci-dessus.
  - Observer la contradiction . . .

Calculablité sur $\mathbb{N}$?

Il n'est peut-être pas évident que des résultats
concernant la calculablité sur $\mathbb{N}$ sont assez puissants,
surtout lorsqu'on les compare avec la généralité des MdT
qui sont capables de travailler avec n' importe quel alphabet fini.

Mais, on peut tout encoder dans $\mathbb{N}$ . . .

Récursion primitive

Nous avons vu que
toutes les fonctions totales ne sont pas primitives-récursives.

Quel est alors l'utilité de ce concept ?

Il existe une relation
avec un langage de programmation, simple et intuitif.

Programmer avec $\mathbb{N}$

Le langage est basé sur un ensemble $\mathbf{Var}$ de ***variables***.

Par $\sigma : \mathbf{Var} \to \mathbb{N}$, nous dénotons un ***environnement*** qui
associe une valeur dans $\mathbb{N}$ à chaque variable dans $\mathbf{Var}$.
Nous dénotons par $\mathbf{Env}$ l'ensemble de tous les environnements.
Exemple: $\mathbf{Var} = \{\, x, y \,\}$ et $\sigma = \{\, (x \mapsto 42), (y \mapsto 0) \,\}$.

Un programme de ce langage va
transformer un environnement initial en un autre.
Pour cela, nous définissons l'***actualisation*** $\sigma[x \leftarrow a]$ d'une
variable x dans un environnement σ par une valeur a :

$$\begin{aligned} \sigma[x \leftarrow a](y) &\stackrel{\text{def}}{=} \sigma(y) \qquad \text{si } y \neq x \\ \sigma[x \leftarrow a](x) &\stackrel{\text{def}}{=} a \end{aligned}$$

Noter que $\sigma[x \leftarrow a] \in \mathbf{Env}$ est un environnement.

Sémantique

Un programme de ce langage va transformer un environnement
initial en un autre . . . *ou il va boucler*.

Ainsi, un programme p
est interprété par une fonction partielle :

$$[\![\, p \,]\!] : \mathbf{Env} \to \mathbf{Env}$$

L'interprétation $[\![\cdot]\!]$ donnera, de manière formelle,
une ***sémantique dénotationelle*** aux programmes.

Exemple:
Si « $x := y + 1$ » est un programme, alors $[\![x := y + 1]\!]$ est une
fonction (partielle) travaille sur un environement qui contient des
valeurs pour les variables x et y.

$$[\![x := y + 1]\!](\{x \mapsto 42, y \mapsto 1\}) = \{x \mapsto 2, y \mapsto 1\}$$

Langages de programmation For et While

La ***syntaxe*** des langages est donnée par induction sur les
constructions suivantes où les métavariables x, y représentent
les variables, p, q les programmes, et $\otimes$ les éléments de
$\{\, >, \geq, \leq, <, =, \neq \,\}$:

(i)	*affectation*	$x := 0 \quad x := y + 1 \quad x := y$
(ii)	*composition sequentielle*	$p \; ; \; q$
(iii)	*condition*	$\mathbf{if}\ x \otimes y\ \mathbf{then}\ p\ \mathbf{else}\ q$
(iv)	*boucle déterminée*	$\mathbf{for}\ y\ \mathbf{do}\ p$
(v)	*boucle indéterminée*	$\mathbf{while}\ x \otimes y\ \mathbf{do}\ p$

La ***sémantique intuitive*** de ces langages est assez simple.
Estimez les résultats de ces deux programmes :

$$x := 0 \; ; \; \mathbf{for}\ y\ \mathbf{do}\ x := x + 1$$
$$x := 0 \; ; \; \mathbf{while}\ x = x\ \mathbf{do}\ x := x + 1$$

Sémantique formelle du langage For

$$
\begin{array}{rcl}
[\![x := 0]\!](\sigma) & \stackrel{\text{def}}{=} & \sigma[x \leftarrow 0] \\
[\![x := y]\!](\sigma) & \stackrel{\text{def}}{=} & \sigma[x \leftarrow \sigma(y)] \\
[\![x := y+1]\!](\sigma) & \stackrel{\text{def}}{=} & \sigma[x \leftarrow \sigma(y)+1] \\
\\
[\![p\,;q]\!](\sigma) & \stackrel{\text{def}}{=} & \Big([\![q]\!] \circ [\![p]\!]\Big)(\sigma) = [\![q]\!]\Big([\![p]\!](\sigma)\Big) \\
\\
[\![\textbf{if } x \otimes y \textbf{ then } p \textbf{ else } q]\!] & \stackrel{\text{def}}{=} & \begin{cases} [\![p]\!](\sigma) & \text{si } \sigma(x)\ [\![\otimes]\!]\ \sigma(y) \\ [\![q]\!](\sigma) & \text{sinon} \end{cases} \\
\\
[\![p]\!]^0(\sigma) & \stackrel{\text{def}}{=} & \sigma \\
[\![p]\!]^{n+1}(\sigma) & \stackrel{\text{def}}{=} & [\![p]\!]\Big([\![p]\!]^n(\sigma)\Big) \\
[\![\textbf{for } y \textbf{ do } p]\!] & \stackrel{\text{def}}{=} & [\![p]\!]^{\sigma(y)}
\end{array}
$$

Sémantique formelle du langage While

$[\![\textbf{while } x \otimes y \textbf{ do } p]\!]$

$$
\stackrel{\text{def}}{=} \begin{cases} [\![p]\!]^n(\sigma) & \text{si } n \text{ est le plus petit nombre tel que} \\ & \quad 1.\ \sigma_n \stackrel{\text{def}}{=} [\![p]\!]^n(\sigma) = \downarrow \ \text{(c-à-d, est défini)} \\ & \quad 2.\ \neg\Big(\ \sigma_n(x)\ [\![\otimes]\!]\ \sigma_n(y)\ \Big) \\ \\ \uparrow & \text{si un tel } n \text{ n'existe pas} \end{cases}
$$

Comparer :

$$
\begin{array}{rcl}
q_1 & = & [\![\textbf{for } y \textbf{ do } p]\!] \\
q_2 & = & [\![z := 0\,;\, w := y\,;\, \textbf{while } z < w \textbf{ do } \big(\ p\,;\, z := z+1\ \big)]\!]
\end{array}
$$

Théorème

1. Pour toute fonction μ-réc. (resp. pr.-réc.) $f : \mathbb{N}^n \to \mathbb{N}$
   il existe un programme p en While (resp. For)
   et $\{ x_0, x_1, \ldots, x_n \} \subseteq \mathbf{Var}$ tel que
   pour tout environnement $\sigma : \mathbf{Var} \to \mathbb{N}$:
   - ☐ $[\![p]\!](\sigma) = \downarrow$ ssi $f(\sigma(x_1), \ldots, \sigma(x_n)) = \downarrow$, et
   - ☐ si c'est le cas, alors $\Big([\![p]\!](\sigma)\Big)(x_0) = f(\sigma(x_1), \ldots, \sigma(x_n))$
2. Pour tout programme p en While (resp. For)
   n'utilisant que les variables $\{ x_1, \ldots, x_n \} \subseteq \mathbf{Var}$,
   il existe $\forall 1 \leq i \leq n$ des fonctions μ-réc. (resp. pr.-réc.)
   $f_i : \mathbb{N}^n \to \mathbb{N}$ tel que pour tout environnement $\sigma : \mathbf{Var} \to \mathbb{N}$:
   - ☐ $[\![p]\!](\sigma) = \downarrow$ ssi $\forall 1 \leq i \leq n : f_i(\sigma(x_1), \ldots, \sigma(x_n)) = \downarrow$, et
   - ☐ si c'est le cas, alors
     $\forall 1 \leq i \leq n : \quad \Big([\![p]\!](\sigma)\Big)(x_i) = f_i(\sigma(x_1), \ldots, \sigma(x_n))$

Styles de programmation

- ☐ Les programmes While sont ***impératifs*** :
  ils manipulent une mémoire, l'environement σ de variables,
  en exécutant des effets de bords
  réalisés par les affectations aux variables.

  Noter qu'il faut, dans le Théorème 1 du transparent précédent, introduire une variable x_0 qui, par définition, doit à la fin d'un calcul contenir le résultat du calcul.
- ☐ Les « programmes » représentés par les fonctions composées de manière récursive sont ***fonctionnels*** :
  ils prennent des arguments d'entrée et
  ils retournent une seule valeur comme résultat.
- ☐ Néanmoins, la construction des fonctions récursives est contrainte de façon à pouvoir construire des nouvelles fonctions que par des schémas de composition, récursion primitive, et minimisation.

Programmation fonctionnelle

- ☐ En généralisant le mécanisme de composition des fonctions — dîtes d'***ordre supérieur*** — il est possible de simplifier le formalisme de façon à ce que l'on n'aie même plus besoin des fonctions de bases et de projection !
- ☐ Cette généralisation se fait comme suit :
  toute fonction prend comme arguments des autres fonctions, et retourne comme résultat . . . une fonction.
- ☐ Le formalisme résultant est le λ-***calcul*** (Church, 1936)
  Il est à la base de la programmation fonctionnelle.
  LISP, Scheme, Miranda, ML, Haskell, Scala . . .

λ-calcul

Une notation pour représenter les fonctions.

Exemple: La fonction $f : \mathbb{N}^2 \to \mathbb{N}$

$$f(x, y) = x + y$$

s'écrit avec la notation λ :

$$f = \underbrace{\lambda}_{\text{lieur}}\ \underbrace{x\,y}_{\text{variables}}\ .\underbrace{x+y}_{\text{corps}}$$

Quelle est la différence ?

La notation construit des ***fonctions anonymes*** ($\lambda x\,y.x+y$)
auxquelles on peut, si l'on souhaite, associer ($=$) un nom (f).

Noter que la fonction anonyme $\lambda x\,y.x+y$
existe aussi sans être associée à un nom.

λ-calcul pur

La ***syntaxe*** du λ-calcul pur est donnée par induction sur les constructions suivantes où la métavariable x représente les variables, et M, N les λ-termes.
L'ensemble $\mathcal{L}$ des λ-termes est alors défini par :

$$\frac{}{x \in \mathcal{L}} \qquad \frac{M \in \mathcal{L}}{\lambda x.M \in \mathcal{L}} \qquad \frac{M \in \mathcal{L} \qquad N \in \mathcal{L}}{MN \in \mathcal{L}}$$

Un terme de la forme $\lambda x.M$ est une λ-***abstraction***.
Un terme de la forme MN est une ***application***.
Attention, l'application n'est pas associative : $MNP = (MN)P$.
(Nous utilisons des parenthèses pour clarifier la structure . . .)

La ***sémantique intuitive*** est qu'un terme de la forme $(\lambda x.M)N$ peut être évalué par la ***substitution*** $[^N/_x]M$ des occurrences de la variable x dans M par N.

Sémantique opérationelle du λ-calcul (I)

La relation $\to \subseteq \mathcal{L} \times \mathcal{L}$ de ***réduction*** sur $\mathcal{L}$ est défini par :

$$(\beta)\ \frac{}{(\lambda x.M)N \to [^N/_x]M} \qquad \frac{N \to N'}{NM \to N'M} \qquad \frac{M \to M'}{NM \to NM'}$$

Exemple:

1. $(\lambda x.x)z \to ?$
2. $z(\lambda x.x) \to ?$
3. $(\lambda x.x)(\lambda z.z) \to ?$
4. $(\lambda x.(\lambda y.xy))(\lambda z.z) \to ?$
5. $(\lambda x.(\lambda y.xy))(\lambda z.z)(\lambda w.w) \to ?$
6. $(\lambda x.(\lambda y.xy))(\lambda z.y)(\lambda w.w) \to ?$

Sémantique opérationelle du λ-calcul (II)

Quelle est la différence entre

$$\lambda z.zy \quad \text{et} \quad \lambda w.wy \quad \text{et} \quad \lambda y.yw \quad ?$$

Le choix (du nom) d'une variable liée n'est pas important.
Par contre, le choix (du nom) d'une variable non-liée l'est.

Nous définissons la congruence $\equiv_\alpha \subseteq \mathcal{L} \times \mathcal{L}$ telle que $M \equiv_\alpha N$ ssi M et N se distinguent uniquement par un renommage consistant de certaines variables liés.

Exemple:

1. $\lambda x.xwx \not\equiv_\alpha \lambda y.ywx$ $\qquad y(\lambda x.xwx) \not\equiv_\alpha x(\lambda y.ywy)$
2. $\lambda x.xwx \equiv_\alpha \lambda y.ywy$ $\qquad x(\lambda x.xwx) \equiv_\alpha x(\lambda y.ywy)$
3. $\lambda x.x(\lambda x.x)x \not\equiv_\alpha \lambda y.y(\lambda x.y)y$

Sémantique opérationelle du λ-calcul (III)

Nous rajoutons une règle pour définir la relation de réduction :

$$\ldots \qquad (\alpha)\ \frac{M \equiv_\alpha N}{M \to N}$$

En plus, nous supposons qu'une substitution $[^N/_x]M$ est définie que si elle n'introduit pas de nouvelles liaisons de variables.

Exemple:

1. $[^x/_y](\lambda x.zy)$ n'est pas définie.
2. $[^{(\lambda z.x)}/_y](\lambda x.zy)$ n'est pas définie.

Sémantique opérationelle du λ-calcul (IV)

La réduction d'un λ-terme n'est pas déterministe,
mais ***confluente*** :

Proposition:
Si $M \to M_1$ and $M \to M_2$,
alors il existe un M' tel que $M_1 \to M'$ and $M_2 \to M'$.

Définition:
Un terme M est en ***forme normale*** s'il n'a pas de réductions.
Une forme normale correspond à une MdT qui s'est arrêtée.

Proposition:
Il existe des termes qui n'arrivent jamais, par réductions consécutives, à une forme normale.

Exemple: Penser à la diagonalisation ...

Calculabilité

- Les valeurs booléennes peuvent être encodées dans le λ-calcul.

$$\begin{array}{lcl} \textbf{True} & \stackrel{\text{def}}{=} & \lambda x.\lambda y.x \\ \textbf{False} & \stackrel{\text{def}}{=} & \lambda x.\lambda y.y \\ \textbf{if } B \textbf{ then } M \textbf{ else } N & \stackrel{\text{def}}{=} & BMN \end{array}$$

- Les nombres naturels peuvent être encodés dans le λ-calcul.

$$\overline{n} \stackrel{\text{def}}{=} \lambda f.\lambda x.\underbrace{f(f(\cdots(f}_{n \text{ fois}}\, x)\cdots))$$

- Toute fonction récursive peut être exprimée dans le λ-calcul.