Le λ-calcul comme modèle de calculabilité

Pablo Rauzy

Ens, dept info

21 janvier 2010

Le λ-calcul

Qu'est ce que c'est ?

Le lambda-calcul (ou λ-calcul) est un système formel inventé par *Alonzo Church* dans les années 1930, qui fonde les concepts de fonction et d'application.

À quoi ça sert ?

- C'est le premier formalisme utilisé pour définir et caractériser les fonctions récursives, il a donc une grande importance dans la théorie de la calculabilité, à l'égal des machines de Turing.
- Il a été appliqué comme langage de programmation théorique.
- Et comme métalangage pour la démonstration formelle assistée par ordinateur.

Le λ-calcul

Syntaxe

La syntaxe du lambda-calcul est très simple.

Le lambda-calcul définit des entités syntaxiques que l'on appelle des lambda-termes (ou parfois aussi des lambda expressions) et qui se rangent en trois catégories :

- Les *variables* : `x`, `y`, ... sont des lambda-termes ;
- les *applications* : `u v` est un lambda-terme si `u` et `v` sont des lambda termes ;
- les *abstractions* : λ`x.v` est un lambda-terme si `x` est une variable et `v` un lambda-terme.

L'application

L'application peut être vue ainsi : si `u` est une fonction et si `v` est son argument, alors `u v` est le résultat de l'application de la fonction `u` à `v`.

L'abstraction

L'abstraction λ`x.v` peut être interprétée comme la formalisation de la fonction qui, à `x`, associe `v`, où `v` contient en général des occurrences de `x`.

Exemple : la fonction $x \to x + 13$ sera dénotée en lambda-calcul par l'expression λ`x.x+13`. L'application de cette fonction au nombre 29 s'écrit (λ`x.x+13`)`29` et "s'évalue" (ou se *normalise*) en l'expression `29+13`.

Le λ-calcul

Notations

Parenthésage

Deux conventions : parenthésage du terme de tête ou associativité à gauche. Cette dernière me semble bien plus naturelle, on va donc l'utiliser.
La syntaxe du parenthésage est donc :

$$\Lambda ::= \texttt{var} \mid \lambda\ \texttt{var}\ \texttt{'.'}\ \Lambda \mid \Lambda\ \texttt{'('}\ \Lambda\ \texttt{')'}$$

Exemple : l'expression `((a b) (c d))` peut se noter `a b (c d)`.

Curryfication

Une abstraction ne se fait que sur une variable, mais *Shönfinkel* et *Curry* ont introduit la *curryfication* : la fonction qui au couple (x, y) associe u peut être vu comme la fonction qui à x associe une fonction qui, à y, associe u.
Elle se note λ`x.(`λ`y.u)`, λ`x.`λ`y.u`, λ`x`λ`y.u` ou tout simplement λ`xy.u`.

Le λ-calcul

Variables libres, variables liées

En lambda-calcul, une variable est *liée* par un λ. Une variable liée a une portée local et on peut par conséquent la renommer sans changer la valeur de l'expression où elle figure. Une variable qui n'est pas liée est dite libre.

Exemple : Dans l'expression λ`x.xy`, la variable `x` est liée et `y` est libre. On peut réécrire ce terme en λ`t.ty`.

λ`bn.banane` équivaut à λ`pt.patate`. (c'est de moi ! :-p)

Il s'agit de remplacer, dans un terme, une variable par un terme.

Ce mécanisme est à la base de la *réduction* qui est le mécanisme fondamental de l'évaluation des expressions et donc du "calcul" des lambda-termes.

La *substitution* dans un lambda-terme $\mathtt{t}$ d'une variable $\mathtt{x}$ par un terme $\mathtt{u}$ est notée $\mathtt{t[x/u]}$. Afin de ne pas pouvoir lier une variable qui était libre avant la substitution, on définit cette dernière par récurrence sur le terme $\mathtt{t}$:

- Si $\mathtt{t}$ est une variable alors $\mathtt{t[x/u]} = \mathtt{u}$ si $\mathtt{x} = \mathtt{t}$ et $\mathtt{t}$ sinon ;
- si $\mathtt{t} = \mathtt{v\ w}$ alors $\mathtt{t[x/u]} = \mathtt{v[x/u]\ w[x/u]}$;
- si $\mathtt{t} = \lambda\mathtt{y.v}$ alors $\mathtt{t[x/u]} = \lambda\mathtt{y.(v[x/u])}$ si $\mathtt{x} \neq \mathtt{y}$ et $\mathtt{t}$ sinon.

Le λ-calcul

Réduction

La *réduction* est bien ce à quoi on s'attend, c'est à dire "faire" les applications afin de "réduire" les abstractions.

Exemple : la réduction de $(\lambda x.xx)(\lambda y.y)$ donne $(\lambda y.y)(\lambda y.y)$.

On appelle *rédex* un terme de la forme $(\lambda x.u)v$. On définit la bêta-contraction de $(\lambda x.u)v$ comme $u[x/v]$.

On note $\twoheadrightarrow$ la fermeture réflexive transitive de la relation $\rightarrow$ de réduction et $=_\beta$ sa fermeture réflexive symétrique et transitive, appelée bêta-conversion.

La β-conversion permet de faire une "marche arrière" à partir d'un terme. Par exemple, de retrouver le terme avant une β-réduction. Passer de x à $(\lambda y.y)x$. Cela s'appelle une β-expansion.

On peut écrire $M =_\beta M'$ si il existe $N_1,\ldots,N_p$ tels que $M = N_1$, $M' = N_p$ et $N_i \rightarrow N_{i+1}$ ou $N_{i+1} \rightarrow N_i$.

Un lambda-terme $\mathtt{t}$ est dit en forme normale si aucune bêta-contraction ne peut lui être appliqué, c'est-à-dire si $\mathtt{t}$ ne contient aucun rédex.

Dans le cas contraire, on dit que $\mathtt{t}$ est *normalisable*. Si de plus toutes les réductions à partir de $\mathtt{t}$ sont finies, alors on dit que le $\mathtt{t}$ est *fortement normalisable*.

Théorème de Church-Rosser

Soient $\mathtt{t}$ et $\mathtt{u}$ deux termes tels que $\mathtt{t} =_\beta \mathtt{u}$. Il existe un terme $\mathtt{v}$ tel que $\mathtt{t} \twoheadrightarrow \mathtt{v}$ et $\mathtt{u} \twoheadrightarrow \mathtt{v}$.

Théorème du losange (ou de confluence)

Soient $\mathtt{t}$, $\mathtt{u}_1$ et $\mathtt{u}_2$ des lambda-termes tels que $\mathtt{t} \twoheadrightarrow \mathtt{u}_1$ et $\mathtt{t} \twoheadrightarrow \mathtt{u}_2$. Alors il existe un lambda-terme $\mathtt{v}$ tel que $\mathtt{u}_1 \twoheadrightarrow \mathtt{v}$ et $\mathtt{u}_2 \twoheadrightarrow \mathtt{v}$.

La calculabilité

La théorie de la calculabilité est une branche de l'informatique théorique et de la logique mathématique.

La notion intuitive que l'on a est qu'une fonction calculable est une fonction qui peut être définie par un algorithme. C'est à dire une suite finie d'opérations clairement explicables.

La formalisation plus mathématique de la définition passe par un *modèle de calcul* comme les fonctions récursives, les machines de Turing, les automates cellulaires, le lambda-calcul...

La **thèse de Church** affirme que la notion intuitive et la définition mathématique coïncident. Et on peut montrer qu'effectivement les différents modèles mathématiques sont équivalents : l'ensemble des fonctions calculables par machines de Turing est le même que celui des fonctions récursives et du lambda-calcul.

Simulation de machines de Turing en λ-calcul

La première partie de notre preuve d'équivalence entre le lambda-calcul et les machines de Turing va consister à montrer que le lambda-calcul est au moins aussi puissant que les machines de Turing.

Il suffit pour cela de montrer que l'on peut simuler une *Machine de Turing quelconque* en lambda-calcul, mais d'abord, définissons les primitives dont nous aurons besoin.

Construction d'un langage de programmation en λ-calcul

Booléens et test conditionnel

Définitions

- `true` := λ`ab.a`
- `false` := λ`ab.b`
- `if` := λ`cab.(c a b)`

Vérification

```
     if true X Y
→    (λcab.(c a b)) true X Y
→    (λab.(true a b)) X Y
→    true X Y
→    (λab.a) X Y
→    X
```

Construction d'un langage de programmation en λ-calcul

Opérateurs logiques

Définitions

- and := λab.(if a b false)
- or := λab.(if a true b)
- not := λa.(if a false true)

Vérification

```
    and true (not false)
→   and true ((λa.(if a false true)) false)
→   and true (if false true true)
→   and true true
→   (λab.(if a b false)) true true
→   if true true false
→   true
```

Construction d'un langage de programmation en λ-calcul

Une structure de données : la liste chaînée

Définitions

- cons := λabc.(c a b)
- head := λp.(p (λab.a))
- tail := λp.(p (λab.b))
- nil := λa.true
- nilp := λp.(p (λab.false))

Vérification

head (tail (cons X (cons Y nil)))
$\rightarrow$ head (tail (cons X ((λabc.(c a b)) Y nil)))
$\rightarrow$ head (tail (cons X (λc.(c Y nil))))
$\rightarrow$ head (tail (λc.(c X (λz.(z Y nil)))))
$\rightarrow$ head ((λp.(p (λab.b))) (λc.(c X (λz.(z Y nil)))))
$\rightarrow$ head ((λc.(c X (λz.(z Y nil)))) (λab.b))
$\rightarrow$ head ((λab.b) X (λz.(z Y nil))) $\rightarrow$ head (λz.(z Y nil))
$\rightarrow$ (λz.(z Y nil)) (λab.a) $\rightarrow$ (λab.a) Y nil $\rightarrow$ Y

Construction d'un langage de programmation en λ-calcul

Nombres

Il y a plusieurs façons de représenter les nombres en utilisant les listes. Celle que j'ai trouvé la plus simple est la suivante :

Définitions

- `0` := `cons true true`
- `succ` := λ`n.(cons false n)`
- `zerop` := λ`n.(head n)`
- `pred` := λ`n.(tail n)`

On a donc une valeur choisie pour zéro et des fonctions successeur et prédécesseur qui nous permette de représenter tout les entiers naturels.

Avec les prédicats pour la valeur zéro on a très envie de commencer à écrire des fonctions récursives, mais on a un petit problème...

Construction d'un langage de programmation en λ-calcul

Récursivité

Supponsons qu'on veuille écrire une fonction d'addition pour nos nombres. Il est assez naturel d'écrire :

`add := λab.(if (zerop a) b (add (pred a) (succ b)))`

Le problème, c'est que dans sa définition, `add` est une variable libre, on ne sait pas ce que c'est. On veut donc une définition où `add` serait liée.

On introduit donc un lambda-terme `x-rec := λf.X[x/f]` où `X` est le lambda-terme définissant la fonction récursive `x`. Par exemple pour `add` :

`add-rec := λf.(λab.(if (zerop a) b (f (pred a) (succ b))))`

On remarque que `add-rec add = add` tel qu'on l'a défini originalement mais avez `add` liée. `add` est donc un point fixe de `add-rec` définie de cette manière.

Construction d'un langage de programmation en λ-calcul

Y Combinator !

Combinateur de point fixe

Un combinateur de point fixe est une fonction qui permet pour chaque fonction `f` de trouver un `x` tel quel `x = f x`.
Un combinateur de point fixe simple, et le plus connu, est celui de Curry :

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

Vérification de `Y f = f (Y f)`

$$\begin{aligned}
& \texttt{Y g} \\
\rightarrow\ & \texttt{(}\lambda\texttt{f.((}\lambda\texttt{x.f(x x)) (}\lambda\texttt{x.f(x x)))) g} \\
\rightarrow\ & \texttt{(}\lambda\texttt{x.g(x x)) (}\lambda\texttt{x.g(x x))} \\
\rightarrow\ & \texttt{g ((}\lambda\texttt{x.g(x x)) (}\lambda\texttt{x.g(x x)))} \\
\rightarrow\ & \texttt{g (}\lambda\texttt{f.((}\lambda\texttt{x.f(x x)) (}\lambda\texttt{x.f(x x))) g)}
\end{aligned}$$

Construction d'un langage de programmation en λ-calcul

Récursivité 2

On peut donc maintenant définir les fonctions récursives et on a l'addition avec `Y add-rec`.

Exemple : `(Y add-rec) (succ 0) (succ 0) = succ (succ 0)`.

Construction d'un langage de programmation en λ-calcul

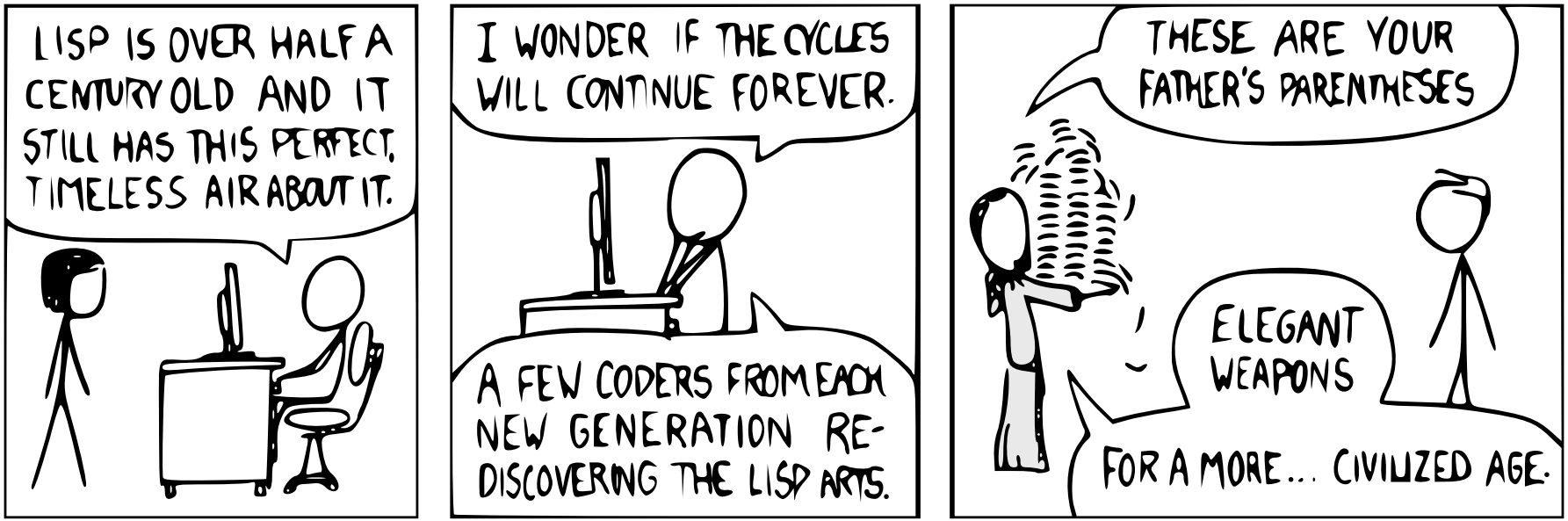
Let's have some fun

```
map := Y (λm.(λfl.
  (if (nilp l)
    nil
    (cons (f (head l)) (m f (tail l))))))

reduce := Y (λr.(λfxl.
  (if (nilp l)
    x
    (r f (f x (head l)) (tail l)))))

filter := Y (λf.(λpl.
  (if (nilp l)
    nil
    (if (p (head l))
      (cons (head l) (f p (tail l)))
      (f p (tail l))))))
```

Construction d'un langage de programmation en λ-calcul



"I've just received words that the Emperor has disolved the MIT Computer Science program permanently."

"Lisp Cycles" - Creative Commons by-nc 2.5 - xkcd.com/297

Codage et simulation de machines de Turing

On choisi une machine de Turing avec un ruban à une bande infinie des deux côtés, et une tête de lecture/écriture.

Voici ses propriétés :

- L'alphabet : `0` et `1` plus un symbole blanc représenté par `nil` ;
- n états représentés par les entiers de 0 à $n - 1$;
- on prend `0` comme état initial ;
- on stocke les états finaux par leur numéro par une liste chaînée.

Il nous reste à coder la fonction de transition et la machine elle même.

Codage et simulation de machines de Turing

Représentation du ruban

On représente le ruban en utilisant les listes chaînées.

Le premier élément contiendra le contenu de la case sur laquelle est la tête de lecture/écriture de la machine.

Le second élément sera une liste spéciale :

- Son premier élément sera lui même une liste chaînée, représentant ce qu'il y a à gauche sur le ruban.
- Son second élément sera la suite de la liste, représentant ce qu'il y a à droite dur le ruban.

On termine ces deux dernières listes par `nil` à la place d'un nouveau constructeur. On en rajoutera un contenant le symbole blanc si on se dépace dessus pour simuler l'infinité du ruban.

Codage et simulation de machines de Turing

Représentation du ruban : exemple

Le ruban vide peut se représenter simplement par
`cons nil (cons nil nil)`.

Si le ruban est le suivant (où # est le caractère blanc) :

```
..., #, 0, 5, 1, 3, 4, #, 2, ...
                ^
```

La tête de lecture étant sur le `1` on représente le ruban par :

```
tape := cons 1 (cons
          (cons 5 (cons 0 (cons nil nil)))
          (cons 3 (cons 4 (cons nil (cons 2 nil)))))
```

Codage et simulation de machines de Turing

Lecture et écriture sur le ruban

La lecture de la case sous la tête de lecture se fait de manière triviale :

`read := λt.(head t)`

On récupère simplement le `head` du ruban `t` (“tape”).

Comme pour la lecture, l'écriture est très simple :

`write := λtv.(cons v (tail t))`

On retourne un ruban construit avec la valeur à écrire `v` et le reste du ruban `t`.

On définit au passage des accesseurs pour la gauche et la droite du ruban :

```
tape-left  := λt.(head (tail t))
tape-right := λt.(tail (tail t))
```

Codage et simulation de machines de Turing

Déplacements de la tête de lecture/écriture

```
shift-head-left := λt.
  (if (nilp (tape-left t))
    (cons nil (cons
      nil
      (cons (read t) (tape-right t))))
    (cons (head (tape-left t)) (cons
      (tail (tape-left t))
      (cons (read t) (tape-right t)))))

shift-head-right := λt.
  (if (nilp (tape-right t))
    (cons nil (cons
      (cons (read t) (tape-left t))
      nil))
    (cons (head (tape-right t)) (cons
      (cons (read t) (tape-left t))
      (tail (tape-right t)))))
```

Codage et simulation de machines de Turing

La fonction de transition

La fonction de transitions doit être de la forme
(état, symbol lu) $\rightarrow$ (état, symbol à écrire, direction de déplacement).

Stratégie : coder une table de transition et une fonction qui la parcourt pour renvoyer les valeurs correspondantes à ses arguments.

On choisi de représenter la table de transition de la manière suivante :

- Une liste dans laquelle le $i^{\text{ème}}$ élément correspondra à l'état `i` ;
- chaque éléments de cette liste sera lui-même une liste de trois éléments correspondants aux valeurs pouvant être lu sur la bande, `nil`, `0` et `1`, dans cet ordre ;
- à leur tour, chacun de ces éléments sera une liste à trois éléments : le symbole à écrire, le nouvel état et la direction dans laquelle aller, codée par `true` pour la gauche et `false` pour la droite.

```
transition := Y (λf.(λTsc.
  (if (not (zerop s))
    (f (tail T) (pred s) c)
    (if (nilp c)
      (head (head T))
      (if (zerop c)
        (head (tail (head T)))
        (head (tail (tail (head T)))))))))
```

Cette fonction renvoie une liste de trois éléments dont on aura besoin donc définissons des accesseurs pour ces éléments :

```
new-symbol := λt.(head t)
next-state := λt.(head (tail t))
direction  := λt.(head (tail (tail t)))
```

Codage et simulation de machines de Turing

Codage de la machine, enfin ! (presque)

Avant de coder la machine on va avoir besoin d'un prédicat d'égalité pour tester si un état est final à l'aide de la fonction `filter` définit précédemment.

```
eqp := Y (λe.(λab.
  (if (and (zerop a) (zerop b))
    true
    (if (or (zerop a) (zerop b))
      false
      (e (pred a) (pred b))))))
```

Codage et simulation de machines de Turing

Codage de la machine

```
simul-mt := Y (λM.(λTFst.
  (if (not (nilp (filter (eqp s) F)))
    t
    ((λf.
      (if (nilp f)
        nil
        (M T F (next-state f)
          (if (direction f)
            (shift-head-left (write t (new-symbol f)))
            (shift-head-right (write t (new-symbol f)))))))
    (transition T s (read t))))))
```

Pour avoir la fonction de simulation d'une machine de Turing particulière `mtp` il faudra coder sa table de transition `trans` et la liste de ses états finaux `fin` puis il suffira de déclarer `mtp := simul-mt trans fin 0`.

On utilise la convention $-B = \bar{B} + 1$.

Le tableau de transition est le suivant :

états ↓ - symbole lu →	0	1	nil
0	(1, 0, false)	(0, 0, false)	(nil, 1, true)
1	(1, 2, true)	(0, 1, true)	(nil, 3, false)
2	(0, 2, true)	(1, 2, true)	(nil, 3, false)
3	-	-	-

Il y a un seul état final : 3.

Codage et simulation de machines de Turing

Exemple : codage en lambda calcul

La table de transition en lambda-calcul :

```
trans := λcqb.(cons c (cons q (cons b nil)))

trans-table := (cons
  (cons (trans nil 1 true) (cons (trans 1 0 false)
                                 (cons (trans 0 0 false) nil)))
  (cons
    (cons (trans nil 3 false) (cons (trans 1 2 true)
                                    (cons (trans 0 1 true) nil)))
    (cons
      (cons (trans nil 3 false) (cons (trans 0 2 true)
                                      (cons (trans 1 2 true) nil)))
      (cons
        (cons nil (cons nil (cons nil nil)))
        nil))))
```

Et la liste des états finaux : `final-state := cons 3 nil.`

Codage et simulation de machines de Turing

Exemple : codage en lambda calcul 2

On peut donc maintenant avoir la machine `opposite` :

```
opposite := simul-mt trans-table final-state 0
```

Si on la lance sur un ruban où il y a un octet signé $_2$00101010 :

```
opposite (cons 0 (cons
  nil
  (cons 0 (cons 1 (cons 0 (cons 1 (cons 0 (cons 1
                                    (cons 0 nil)))))))))
```

Après beaucoup (beaucoup) de réductions cela retournera :

```
cons 1 (cons
  nil
  (cons 1 (cons 0 (cons 1 (cons 0 (cons 1 (cons 1
                                    (cons 0 nil)))))))))
```

C'est à dire $_2$11010110 ce qui est bien ce qu'on attendait.

Simulation du λ-calcul par une machine de Turing

On sait maintenant que le lambda-calcul est au moins aussi puissant que les machines de Turing.

Pour montrer l'équivalence entre les deux modèles on va maintenant montrer qu'il est aussi possible de simuler le lambda-calcul avec une machine de Turing.

Pour cela il nous faut une machine de Turing qui sait *calculer* un lambda-terme, c'est à dire le *réduire*.

Simulation du λ-calcul par une machine de Turing

L'alphabet : les variables

Problème

Une machine de Turing est décrite par un nombre finie de symboles, mais le lambda-calcul contient potentiellement un nombre infini de variable.

Solution

- On se place dans le cas du lambda-calcul sans curryfication,
- on code les variables en binaire par leur index de DeBruijn (profondeur de la variable dans son champ de portée).

C'est bien une solution au problème car cela rend unique la notation des lambda-termes dans une même classe d'équivalence par α-conversion.

Simulation du λ-calcul par une machine de Turing

L'alphabet : les applications et l'encodage

Application

On va noter les applications de manières préfixes, cela sera plus simple de savoir à l'avance ce qu'il se passe lors de la lecture sur le ruban de la machine.

L'encodage

On utilise donc un alphabet $A = \{\lambda, @, v, 0, 1\}$ et on code chaque lambda-terme `M` en son code `#M` de la manière suivante : on note les applications par un '@' avant les deux lambda-termes de l'application, les variables libres par 'v' et les variables liées par 'v' suivi de leur index de DeBruijn en binaire.

Exemple : le lambda-terme `(λa.ab)(λa.λb.λc.acb)` se code de manière unique en : `@λ@v0vλλλ@@v10v0v1`.

Simulation du λ-calcul par une machine de Turing

Le ruban

Le ruban de notre machine de Turing sera constitué de six bandes, avec une tête de lecture/écriture indépendante pour chacune d'elle.

La première bande est celle sur laquelle est entré le lambda-terme à réduire.

Les autres bandes sont nommées ainsi dans la suite :

- pre-rédex ;
- fonction ;
- argument ;
- post-rédex ;
- réduction.

Simulation du λ-calcul par une machine de Turing

Description de la machine

À chaque cycle d'exécution, la machine opère en quatre étapes :

1. La machine lit la première bande en repérant un rédex du lambda-terme qui s'y trouve,
   - la partie fonctionnelle du rédex est écrite dans la bande "fonction",
   - son argument dans la bande "argument",
   - tout ce qui apparaît avant (resp. après) le rédex est écrit dans "pré-rédex" (resp. "post-rédex"),

   s'il n'y a aucun rédex dans le lambda-terme, alors la machine s'arrête.
2. La machine copie le contenu de "fonction" dans "réduction" en omettant le λ initial et en replaçant chaque occurence de la variable liée par le contenu de "argument".
3. La machine remplace la contenu de la première bande par la concaténation de "pré-rédex", "réduction" et "post-rédex", dans cet ordre.
4. La machine efface le contenu de toutes les bandes sauf la première.

Simulation du λ-calcul par une machine de Turing

Exemple : réduction de λa.((λcd.(a e d)) (λg.g)) (λab.a)

On va réduire le lambda-terme λa.((λcd.(a e d)) (λg.g)) (λab.a) à l'aide de la machine de Turing que l'on viens de coder. On va détailler le premier cycle puis on donnera juste les états de réductions successives.

Avant cela, il faut encoder ce lambda-terme dans l'alphabet de notre machine :

λa.((λcd.(a e d)) (λg.g)) (λab.a)
= λa.((λc.λd.(a e d)) (λg.g)) (λa.λb.a)
= @λ@$\lambda\lambda$@@v11vv0λv0$\lambda\lambda$v1

Simulation du λ-calcul par une machine de Turing

Exemple : réduction de `λa.((λcd.(a e d)) (λg.g)) (λab.a)`

Maintenant, voyons comment se déroule le premier cycle :

	première	pre-rédex	fonction	argument	réduction
1	@λ@λλ@@v11vv0λv0λλv1	@	λ@λλ@@v11vv0λv0	λλv1	
2	@λ@λλ@@v11vv0λv0λλv1	@	λ@λλ@@v11vv0λv0	λλv1	@λλ@@λλv1vv0λv0
3	@@λλ@@λλv1vv0λv0	@	λ@λλ@@v11vv0λv0	λλv1	@λλ@@λλv1vv0λv0
4	@@λλ@@λλv1vv0λv0				

La bande “post-rédex” n’est pas représentée car on ne l’utilise pas (et puis y avais pas la place).

Simulation du λ-calcul par une machine de Turing

Exemple : réduction de `λa.((λcd.(a e d)) (λg.g)) (λab.a)`

Et les cycles suivants :

codage machine	signification
`@λλ@@λλv1vv0λv0`	$(\lambda c.\lambda d.((\lambda a.\lambda b.a)\ e\ d))\ (\lambda g.g)$
`λ@@λλv1vv0`	$\lambda d.((\lambda a.\lambda b.a)\ e\ d)$
`@λ@λvv0`	$\lambda d.((\lambda b.e)\ d)$
`λv`	$\lambda d.e$

Il n'y a plus de rédex donc la machine s'arrête et on a bien réduit au maximum le lambda-terme que l'on avait en entrée.

Conclusion

On a donc bien pu simuler les machines de Turing en lambda-calcul, et on est capable de réduire un lambda-terme à l'aide d'une machine de Turing.

On en conclu que les deux modèles sont bien équivalents en terme de calculabilité.

Comme on "sait" / "admet" que les machines de Turing sont un bon modèle de calculabilité, le lambda-calcul en est donc un aussi.

The end

Questions ?