Le λ-calcul comme modèle de calculabilité

Pablo Rauzy

20 janvier 2010

Résumé

Le but de cet article est de montrer que le lambda-calcul est un bon modèle de calculabilité. Pour cela nous allons d'abord aborder les notions de lambda-calcul et de calculabilité dans une présentation brève de chacun de ces deux domaines puis nous montrerons que le lambda-calcul est équivalent aux machines de Turing.

Table des matières

1 Introduction au lambda-calcul

Une bonne partie de cette section provient presque directement de Wikipedia [1].

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. Il a été le premier formalisme utilisé pour définir et caractériser les fonctions récursives et donc il a une grande importance dans la théorie de la calculabilité, à l'égal des machines de Turing. Il a depuis été appliqué comme langage de programmation théorique et comme métalangage pour la démonstration formelle assistée par ordinateur. Le lambda-calcul peut être ou non *typé*. On ne va s'intéresser ici qu'au lambda-calcul non typé.

1.1 Syntaxe

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* 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* 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`.

Ainsi, la fonction qui prend en paramètre le lambda-terme `x` et lui ajoute 13 (c'est-à-dire en notation mathématique courante 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`.

1.2 Notations, conventions et concepts

1.2.1 Parenthésage

On utilise les parenthèses pour délimiter les applications, mais pour ne pas surcharger les notations, on ne met que les parenthèses "utiles". Ainsi l'expression $\mathtt{x}_1\ \mathtt{x}_2\ \ldots\ \mathtt{x}_n$ est équivalente à $(\ldots(\mathtt{x}_1\ \mathtt{x}_2)\ \ldots\ \mathtt{x}_n)$.

Il y a deux conventions de parenthésage, le parenthésage du terme de tête et l'associativité à gauche, que je trouve bien plus naturelle et que nous allons donc utiliser ici. La syntaxe est donc :

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

Exemple : l'expression `((a b) (c d))` se note simplement `a b (c d)`.

1.2.2 Curryfication

Un lambda-terme ne prend qu'un seul argument, mais Shönfinkel et Curry ont introduit la *curryfication* et montré qu'on peut ainsi contourner cette restriction de la façon suivante : la fonction qui au couple (x,y) associe u est considérée comme une fonction qui, à x, associe une fonction qui, à y, associe u. Elle est donc notée : λ`x.(`λ`y.u)`. Cela s'écrit aussi λ`x.`λ`y.u` ou λ`x`λ`y.u` ou tout simplement λ`xy.u`. Par exemple, la fonction qui, au couple (x,y) associe x+y sera notée λ`xy.x+y`.

1.2.3 Variables libres et 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 λx.xy, la variable x est liée et y est libre. On peut réécrire ce terme en λt.ty. λbn.banane équivaut à λpt.patate.

1.3 Substitution et α-conversion

L'outil le plus important pour le lambda-calcul est la *substitution* qui permet 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 t d'une variable x par un terme u est notée 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 t dans lequel on l'applique de la façon suivante :

- si t est une variable alors t[x/u] = u si x = t et t sinon
- si t = v w alors t[x/u] = v[x/u] w[x/u]
- si t = λy.v alors t[x/u] = λy.(v[x/u]) si x $\neq$ y et t sinon

Remarque : dans le dernier cas on fera attention à ce que y ne soit pas une variable libre de u. En effet, elle serait alors "capturée" par le lambda externe. Si c'est le cas on renomme y et toutes ses occurrences dans v par une variable z qui n'apparaît ni dans t ni dans u.

L'α-conversion établit une relation d'équivalence entre lambda-termes. Deux lambda-termes qui ne diffèrent que par un renommage (sans capture nécessaire) sont dit α-convertibles.

1.4 Réductions

Une manière de voir les termes du lambda-calcul consiste à les concevoir comme des arbres ayant des nœuds binaires (les applications), des nœuds unaires (les λ-abstractions) et des feuilles (les variables). Les *réductions* ont pour but de modifier les termes, ou les arbres si on les voit ainsi ; par exemple, la réduction de (λx.xx)(λy.y) donne (λy.y)(λy.y).

On appelle *rédex* un terme de la forme (λx.u)v. On définit la bêta-contraction (ou β-contraction) de (λx.u)v comme u[x/v].

Exemple : (λx.xy)a $\rightarrow$ (xy)[x/a] = ay

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

La β-conversion permet de faire une "marche arrière" à partir d'un terme. Cela permet, par exemple, de retrouver le terme avant une β-réduction. Passer de x à (λy.y)x.

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

Cela signifie que dans une conversion on peut appliquer des réductions ou des relations inverses des réductions (appelées expansions).

On définit également une autre opération, appelée êta-réduction (ou son inverse la êta-expansion), définie ainsi : λx.ux $\rightarrow_\eta$ u, lorsque x n'apparaît pas libre dans u. En effet, ux s'interprète comme l'image de x par la fonction u. Ainsi, λx.ux s'interprète alors comme la fonction qui, à x, associe l'image de x par u, donc comme la fonction u elle-même.

1.5 Normalisation

Le calcul associé à un lambda-terme est la suite de réductions qu'il engendre. Le terme est la description du calcul et la *forme normale* du terme (si elle existe) en est le résultat.

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

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

Exemples : $(\lambda x.x)((\lambda y.y)z)$ est fortement normalisable.
$(\lambda x.xxx)(\lambda x.xxx)$ ne fait que créer des termes de plus en plus grand.

Théorème de Church-Rosser : soient t et u deux termes tels que $t =_\beta u$. Il existe un terme v tel que $t \twoheadrightarrow v$ et $u \twoheadrightarrow v$.

Théorème du losange (ou de confluence) : soient t, u_1 et u_2 des lambda-termes tels que $t \twoheadrightarrow u_1$ et $t \twoheadrightarrow u_2$. Alors il existe un lambda-terme v tel que $u_1 \twoheadrightarrow v$ et $u_2 \twoheadrightarrow v$.

Grâce au Théorème de Church-Rosser on peut facilement montrer l'unicité de la forme normale ainsi que la cohérence du lambda-calcul (c'est-à-dire qu'il existe au moins deux termes distincts non bêta-convertibles).

2 Définition de la calculabilité

Cette section a pour principale référence Wikipedia [2]

La théorie de la calculabilité est une branche de l'informatique théorique et de la logique mathématique. La notion intuitive de fonction calculable est très vielle mais sa formalisation ne date que de la première moitié du siècle dernier (années 1930). La compréhension de ce qui est calculable et de ce qui ne l'est pas permet entre autre de voir les limites de ce que peut résoudre un ordinateur.

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.

C'est comme ça que nous allons montrer que le lambda-calcul est un bon modèle de calculabilité, en montrant son équivalence avec les machines de Turing.

3 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.

3.1 Construction d'un langage de programmation simple à partir du λ-calcul

Cette section s'inspire en partie de Wikipedia et de travaux trouvés sur le net [1, 4, 5]

3.1.1 Condition et valeurs booléennes

Tout d'abord, nous aurons besoin des booléens et d'un test conditionnel. On peut les définir ainsi en lambda-calcul :

- `true  := λab.a`
- `false := λab.b`
- `if    := λcab.(c a b)`

Vérifions que ces choix sont corrects en réduisant l'expression `if true X Y`.

```
     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
```

Ce qui est bien ce à quoi on s'attend. Il est trivial de voir que si on avait mis `false` à la palce de `true` la réduction se serait finie sur `Y`.

Remarque : tel qu'on a défini les booléens `true` et `false`, le `if` ne "sert à rien" : `(if a b c) = (a b c)` et ce sont les valeurs booléennes qui sont en faite des fonctions qui renvoient leur premier (pour `true`) ou leur second (pour `false`) argument.

3.1.2 Opérateurs logiques

On peut maintenant utiliser les valeurs booléennes et le test conditionnel que l'on vient de voir pour définir un **et**, un **ou** et un **non** en lambda-calcul :

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

Effectuons les bêta-réductions de deux expressions booléennes en guise de vérifications rapides :

```
     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
```

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

C'est bien ce qu'on voulait.

3.1.3 Une structure de données : la liste chaînée

Définnissons les listes chaînées. Pour cela on a besoin d'un *constructeur* qui prend deux éléments et en fait une paire, et de deux *accesseurs* respectivement pour le premier et le second éléments d'une paire. Une liste chaînée est simplement une paire dont le premier éléments (la *tête*) est un élément de la liste et le second (la *queue*) la suite de la liste (une autre paire), ou une valeure *nulle* (fin de la liste). On représente la liste vide par la valeur *nulle*. Il nous faut donc aussi un prédicat pour cette valeure.

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

Remarque : `cons`, `head` et `tail` sont construit sur la même structure que le test conditionnel `if`, `true` et `false`. En effet, une paire construite est un lambda-terme de la forme `λp.(p H T)` ou `H` et `T` sont respectivement le premier et le second élément de la paire. On a déjà vu que pour récupérer `H` (resp. `T`) dans le lambda-terme `H T` il suffit de mettre `true` (resp. `false`) devant : c'est exactement ce que font `head` et `tail`.

Encore une fois, effectuons une petite vérification que tout cela fonctionne comme on le souhaite :

```
     head (tail (cons X (cons Y nil)))
→    head (tail (cons X ((λabc.(c a b)) Y nil)))
→    head (tail (cons X (λc.(c Y nil))))
→    head (tail ((λabc.(c a b)) X (λc.(c Y nil))))
→    head (tail (λc.(c X (λz.(z Y nil))))                    on renomme le second c sinon on le confond avec le nouveau.
→    head ((λp.(p (λab.b))) (λc.(c X (λz.(z Y nil)))))
→    head ((λc.(c X (λz.(z Y nil)))) (λab.b))
→    head ((λab.b) X (λz.(z Y nil)))
→    head (λz.(z Y nil))
→    (λp.(p (λab.a))) (λz.(z Y nil))
→    (λz.(z Y nil)) (λab.a)
→    (λab.a) Y nil
→    Y
```

Ce qui est bien ce à quoi on s'attend : le premier élément de la seconde paire, ou vu autrement le second élément de la liste à deux éléments `[X, Y]`.

3.1.4 Nombres et récursivité

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

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

De cette manière on peut construire récursivement tout les nombres entiers et comme on a un prédicat pour zéro, on peut définir des fonctions récursives.

Un exemple avec la fonction d'*addition* :
`add := λab.(if (zerop a) b (add (pred a) (succ b)))`

En fait ce n'est pas tout à fait vrai, puisque dans cette définition le terme `add` est une variable libre, on ne sait pas ce que c'est. Donc ce qu'on cherche, c'est 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. `add` est donc un point fixe de `add-rec` défini de cette manière.

Il nous suffit donc d'utiliser cette propriété et un *combinateur de point fixe* pour pouvoir ensuite définir des fonctions récursives.

Définition : 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 est celui de Curry :
`Y := λf.((λx.f(x x)) (λx.f(x x)))`
On peut vérifier simplement qu'on a effectivement l'équivalence `Y f = f (Y f)` :

Y g
$\rightarrow$ (λf.((λx.f(x x)) (λx.f(x x)))) g
$\rightarrow$ (λx.g(x x)) (λx.g(x x))
$\rightarrow$ g ((λx.g(x x)) (λx.g(x x)))
$\rightarrow$ g (λf.((λx.f(x x)) (λx.f(x x))) g)

On peut donc maintenant définir les fonctions récursives et on a l'addition avec `Y add-rec`.
Par exemple `(Y add-rec) (succ 0) (succ 0) = succ (succ 0)`. Je ne déroule pas le calcul car cela prendrait énormément de place et serait de toutes manières illisible, mais le cœur y est.

3.1.5 Compléments

Nous n'en avons pas besoin ici mais on aurait pu définir beaucoup d'autres choses pour notre petit langage de progammation. Il est par exemple possible assez simplement de coder une autre structure de donnée comme des arbres binaires.

À l'aide du combinateur de point fixe `Y`, on peut aussi coder des fonctions très utiles sur les listes, comme un `map fct list` (qui retourne une liste constituée des éléments de `list` auxquels on a appliqués `fct`), un `filter p list` qui retourne les éléments de `list` qui satisfont le prédicat `p`, un `reduce fct init list` qui renvoie la valeure calculé par `fct (... (fct (fct init` $\mathtt{e}_1$`)` $\mathtt{e}_2$`) ...)` $\mathtt{e}_n$ où les $\mathtt{e}_i$ sont les éléments de `list` dans l'ordre.

Nous n'aurons pas non plus forcément besoin de ces trois fonctions mais les voici au moins pour le plaisir, codées en utilisant ce que l'on a vu depuis le début :

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

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

On va en fait utiliser `filter` pour coder la simulation d'une machine de Turing, d'ailleurs...

3.2 Codage et simulation de machines de Turing

Je n'ai pas trouvé de document dans lequel ce travail été poussé plus loin que le fait de dire que c'est faisable. J'ai donc réalisé cette partie entièrement par moi même. Il se peut donc que certaines choses ne soit pas optimales, mais par contre tout devrait fonctionner d'après mes vérifications.

Avec ce que l'on a déjà fait jusqu'ici vous devriez être convaincu que le lambda-calcul est Turing-complet, le langage que l'on vient de construire rappelant furieusement un langage fonctionnel "pur" comme Scheme (une implémentation très épurée de Lisp).

Mais ne nous arrêtons pas là et voyons une preuve plus formelle de la Turing-complétion du lambda-calcul : codons une fonction de simulation d'une machine de Turing abstraite (n'importe laquelle) avec le lambda-calcul.

3.2.1 La machine de Turing abstraite

On choisi une machine de Turing avec un ruban à une bande infinie des deux côtés, et une tête de lecture/écriture. Avec les nombres `0` et `succ 0` plus un symbole blanc comme alphabet.

Le symbole blanc sera représenté par `nil` et les nombres entiers par ceux qu'on a codé précédemment. À partir de maintenant on note `1` pour `succ 0`, `2` pour `succ (succ 0)`, `3` pour `succ (succ (succ 0))` etc.

La machine aura $n+1$ états codés par les entiers de `0` à `n`, `0` étant l'état initial.

Parmis ces états certains sont finaux. Il seront dans une liste terminée par `nil`.
Par exemple `final-states := cons 42 (cons 13 (cons 51 nil))`.

Il nous reste encore à coder la fonction de transition puis la machine elle même. Pour cela voyons d'abord comment on va représenter le ruban et les différentes opérations nécessaires.

3.2.2 Le ruban

On va représenter un ruban infini des deux côtés par une liste spéciale. Son premier élément sera celui sur lequel la tête de lecture est actuellement. Son second élément ne contiendra pas de case du ruban, mais en `head` la liste de ce qu'il y a en partant vers la gauche du ruban depuis la tête de lecture/écriture, et en `tail` ce qu'il y a vers la droite sur le ruban.

À l'intérieur de cette liste on code le caractère blanc par `nil` et les cases du ruban sont toujours les `head` des maillons de la liste. Des deux côtés, la zone de travail sur le ruban se termine par un `nil` à la place d'un nouveau constructeur, et quand on voudra se déplacer dessus une case sera ajouté au ruban à sa place (un `cons nil nil`), afin de simuler l'infinité du ruban.

Exemple : 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)))))
```

Les différents chiffres sont là juste pour que l'exemple soit plus explicite. La machine de Turing que l'on va coder n'aura bien pour alphabet que trois symboles `0`, `1` et `nil`.

Remarque : Un ruban complètement vide est simplement codé par `cons nil (cons nil nil)`.

3.2.3 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`.

3.2.4 Déplacements de la tête de lecture/écriture

Les déplacements sont des opérations un peu moins simples, il faut retourner un ruban correspondant au précédent décalé d'une case.

Pour rendre ces fonctions plus lisibles on va en définir avant deux accesseurs `tape-left` et `tape-right` qui renvoie respectivement la gauche et la droite du ruban par rapport à la case courante (non incluse).

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

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)))))
```

Comme on peut le voir, on simule l'infinité du ruban en testant si le ruban du côté vers lequel on veut se déplacer est nul. Si oui on avance en créant une nouvelle case vide (avec `nil` dedans, le caractère blanc). Si non on se déplace normalement, d'une case vers la direction demandée.

3.2.5 La fonction de transition

La stratégie que l'on va adopter pour la fonction de transition est d'encoder une table de transitions avec des listes chainées, et d'avoir une fonction qui parcourt cette table pour effectuer les transitions.

Pour la table des transitions, on a besoin de connaître le symbole à écrire sur la bande, le nouvel état et la direction de déplacement en fonction de l'état actuel et du symbole sous la tête de lecture/écriture.

On va construire une liste dans laquelle le $i^{\text{ème}}$ élément correspondra à l'état `i`. Chacun des 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.

Voyons maintenant comment on va coder la fonction de transition, ses arguments sont `T` la table de transition, `s` l'état et `c` le symbole lu sur la bande :

```
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)))))))))
```

Comme on le remarque facilement à la présence du combinateur de point fixe `Y`, la fonction de transition est une fonction récursive. On parcours la table de transition en décrémentant l'état jusqu'à `0` de manière à arriver dans l'état actuel (du moins celui passé en paramètre), puis une fois dans le bon état on retourne le premier, le second ou le troisième élément de la liste correspondante, selon que le caractère lu soit respectivement `nil`, `0` ou `1`.

La fonction `transition` renvoie donc une liste de la forme `cons c (cons q (cons b nil))` où `c` est le symbole à écrire sur la bande, `q` est le nouvel état et `b` est la direction de déplacement de la tête.

Remarque : aucune gestion des erreurs n'est faite (état inexistant parce que trop grand par exemple). Ce n'est pas l'objet de la démonstration et ne serait en plus pas très compliqué à rajouter, mais cela nuirait à la clarté du code et je préfère donc ne garder que ce qui est utile à la démonstration.

3.2.6 Construction de la machine

Encore une fois, définissons quelques accesseurs pour rendre le code plus lisible. Ils prennent en argument ce qui est retourné par la fonction de transition :

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

Il nous faut aussi un moyen de tester si un état est final ou pas. On a les états finaux dans une liste et une fonction `filter`. Il nous faut donc un prédicat d'égalité numérique :

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

Le prédicat `eqp` prend en argument deux nombres dans le codage qu'on a définit précédemment et vérifie si en les décrémentant simultanément on arrive bien à `0` en même temps.

On peut maintenant coder la fonction qui simule la machine de Turing comme une fonction récursive `simul-mt` qui prend en argument la table de transition `T` de la machine de Turing à simuler, la liste de ses états finaux `F`, son état `s`, et son ruban `t`.

Il suffira de l'appeler avec `0` comme état au départ et la fonction s'arrêtera si la machine de Turing codée dans la table de transition s'arrête. Elle renvoi dans ce cas le ruban dans son état actuel (en vue d'une éventuelle analyse d'un résultat écrit sur le ruban par la machine) si elle est dans un état final, et `nil` si elle est arrêté parce qu'il n'y a pas de transitions existantes dans l'état où elle est (choix arbitraire).

```
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`. En supposant que l'on veuille lancer `mtp` sur un ruban contenant juste un `1` on pourra la simuler avec `mtp (cons 1 (cons nil nil))`.

On a donc bien montré que tout ce que peut calculer une machine de Turing peut-être calculé par le lambda-calcul, puisqu'on est capable de simuler n'importe quelle machine de Turing en lambda-calcul. Pour finir notre preuve il reste à montrer l'autre sens de l'équivalence : que l'on peut simuler n'importe quel calcul réalisable en lambda-calcul (c'est à dire une réduction de lambda-terme) sur une machine de Turing.

Mais avant, voyons un exemple concret de simulation.

3.2.7 Exemple : calcul de l'opposé d'un nombre binaire

On va coder une machine de Turing d'exemple, qui calculera l'opposé d'un nombre en binaire, avec la convention que $-B = \bar{B} + 1$. Le nombre B devra être écrit en binaire sur le ruban avec la tête sur le bit

de signe (le premier bit du nombre écrit de gauche à droite). Le résultat sera $-B$ en binaire avec la tête à nouveau sur le bit de signe.

Voyons déjà la table de transition de cette machine de Turing au format "(symbole à écrire, nouvel état, direction de déplacement)" toujours avec `true` pour gauche et `false` pour droite :

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	-	-	-

Codons maintenant la même table de transition en lambda-calcul, en suivant la convention définit précédemment. Une fois de plus définissons avant une petite fonction `trans` qui allègera le code afin de le rendre plus lisible.

```
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))))
```

Chaque ligne correspond à un état, comme dans la table sauf qu'on suit la convention choisie précédemment de commencer dans chaque état par le cas où on a lu le symbole `nil`, puis `0`, et enfin `1`.

Le seul état final est `3` donc on pose `final-state := cons 3 nil`.

On peut donc avoir notre machine de Turing `opposite` :
`opposite := simul-mt trans-table final-state 0`
puis la lancer sur un ruban avec écrit 42 en binaire sur 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)))))))))
```

Ce qui 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 vaut bien -42 en entier signé sur un octet.

4 Simulation du λ-calcul par une machine de Turing

Cette section a été rédigée après la lecture d'un chapitre d'un polycopié de cours sur le sujet [6]

Afin de compléter notre preuve d'équivalence entre lambda-calcul et machine de Turing, il nous faut montrer que l'on peut construire une machine de Turing qui "interprète" le lambda-calcul. C'est à dire qui est capable de simuler le calcul (la réduction) d'un lambda-terme, autrement dit de faire des bêta-réductions de lambda-termes normalisables.

4.1 L'alphabet

Une machine de Turing est un objet dont la description est finie. Il faut donc un alphabet avec un nombre fini de symbole pour la machine de Turing que l'on veut construire. Le lambda-calcul pouvant avoir une infinité de variable, il faut un moyen de les coder de manière finie.

La solution retenue est de les noter par des chaînes en binaire représentant la profondeur des variables liées dans leur champs de portée. On peut faire cela car pour simplifier les choses on se place dans un contexte de lambda-calcul sans curryfication. Cette façon de numéroter les variables s'appelle l'indexation de DeBruijn[7] et a la propriété de rentre unique la notation des lambda-termes d'une même classe d'équivalence par α-conversion.

On va aussi 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.

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 en : `@λ@v0vλλλ@@v10v0v1`.

4.2 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.

Le lambda-terme en entré est écrit sur la première bande. On appelle les cinq autres bandes dans l'ordre : pre-rédex, fonction, argument, post-rédex et réduction.

4.3 L'exécution

On ne va pas décrire ici formellement la machine de Turing en donnant sa table de transition complète, mais on va décrire les différentes étapes de son algorithme. Le lecteur verra aisément que chacune des étapes/opérations énoncées est tout à fait réalisable par une machine de Turing avec le codage des lambda-termes que l'on vient de décrire.

4.3.1 Description

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.

4.3.2 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 étates 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@λλ@@v11vv0λv0λλv1
```

Maintenant, voyons comment se déroule le premier cycle :

étape	première	pre-rédex	fonction	argument	post-rédex	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					

Et les cycles suivants :

codage machine	signification
@λλ@@λλv1vv0λv0	(λc.λd.((λa.λb.a) e d)) (λg.g)
λ@@λλv1vv0	λd.((λa.λb.a) e d)
@λ@λvv0	λd.((λb.e) d)
λv	λd.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.

4.4 Conclusion

On a donc bien une machine de Turing capable de simuler un calcul de lambda-terme, ce qui fini de montrer l'équivalence entre lambda-calcul et machine de Turing, et par la même occasion de nous convaincre que le lambda-calcul est bien un bon modèle de calculabilité.

Références

[1] Wikipedia, *Lambda calculus*, sous license Creative Commons by-sa
http://en.wikipedia.org/wiki/Lambda_calculus

[2] Wikipedia, *Computability theory*, sous license Creative Commons by-sa
http://en.wikipedia.org/wiki/Computability_theory

[3] Wikipedia, *Turing Machine*
http://en.wikipedia.org/wiki/Turing_Machine

[4] M. J. Dominus, Free Talks, *Perl Contains the Lambda Calculus*
(How to write a 163 lines program to compute 1+1)
http://perl.plover.com/yak/lambda/samples/

[5] xkcd's forums, *The Turing Machine vs. Lambda Calculus* thread
http://forums.xkcd.com/viewtopic.php?f=40&t=46449

[6] U. Dal Lago et S. Martini, *The Weak Lambda Calculus as a Reasonable Machine*, section 5
Dipartimento di Scienze dell'Informazione, Università di Bologna.
http://www.cs.unibo.it/~martini, http://www.cs.unibo.it/~dallago

[7] Wikipedia, *De Bruijn index*
http://en.wikipedia.org/wiki/De_Bruijn_index