Aide de l'interpréteur de Lambda Calcul

Gilbert Dhuime

30 avril 2012

Résumé

Les propriétées spécifiques de cet interprèteur sont :
Un environnement intégré simple.
Une grammaire qui autorise l'écriture d'alias pour les abstractions.
Un prélude constitué par un ensemble prédéfini d'alias.
La possibité de créer et de sauvegarder ses propres alias.
Une prise en main rapide du logiciel en faisant le tour de toutes les fonctions standarts.

1 L'espace de travail

La barre des menus
L'espace de travail est constitué, dans sa partie supérieure, d'une barre de menus déroulants classique : File , Edit , Help.
Le menu File sert à sauver (resp à restituer) l'environnement constitué par les déclarations des alias qui sont visualisées dans la fenêtre historique.
Le menu Edit permet les opérations courantes de couper , coller dans la fenêtre de commandes.
Le menu Help permet d'accéder à l' aide en ligne.
En dessous de cette barre se trouvent à gauche (resp à droite) la fenêtre des commandes (resp la fenêtre de l'historique) .
A la base de ces dernières se trouve la fenêtre des erreurs.
De ces trois fenêtres seule la fenêtre des commandes est éditable.

La fenêtre des commandes
Dans le panneau de Commande s'affiche , au démarrage, le message :
Lambda Interpreter
Version 1.0
Les commandes sont "let", "lterme", "quit" où une application.

La commande let est associée à la définition des alias.
La syntaxe de let est : let alias = abstraction.
Elle crée un nouvel alias qui dénote le L-terme écrit à droite du caractère '='.
Exemple : let cons = Lx.Ly.((pair false) ((pair x) y))
On note que le nouvel alias "cons" se définit à partir des alias prédéfinis "pair" et "false".

La commande quit permet de sortir de l'interpréteur.
La commande lterme permet de visualiser le L-terme associé à un alias.

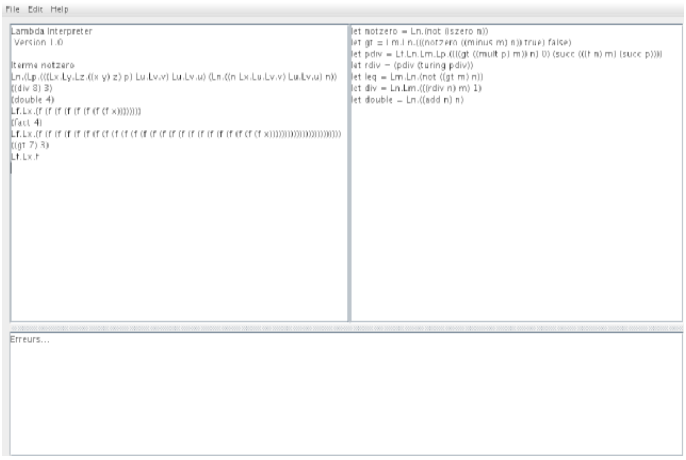

FIGURE 1 – L'espace de travail

Par exemple "lterme cons" renvoie :
Lx.Ly.((Lx.Ly.Lf((f x) y) Lu.Lv.v) (Lx.Ly.Lf((f x) y) x) y)).
On remarque que le L-terme renvoyé par cette commande est sans alias .
Ainsi les alias "pair" et "false" ont été remplaçés par les L-termes qu'ils identifient.

Exécuter une application
Par exemple ((add 3) 4) s'exécute dans la fenêtre des commandes et donne :
Lf.Lx.(f (f (f (f (f (f (f x)))))))).

Une commande s'exécute à l'enfoncement de la touche Enter.
Si la commande est une application le résultat est affiché dans la fenêtre de commandes .
Si une erreur d'origine syntaxique est déterminée par cette commande le message d'erreur est affiché dans la fenêtre des erreurs.

La fenêtre d'historique des alias
Si la commande let s'exécute avec succès l'alias et l'abstraction associée s'affichent dans cette fenêtre, permettant ainsi de visualiser l'état actuel de l'environnement.
Si une erreur de syntaxe est détectée l'erreur s'affiche dans la fenêtre des erreurs.

La fenêtre des erreurs
Dans cette dernière s'affichent les erreurs , le plus souvent de syntaxe.

2 La grammaire

Les L-termes , où termes du lambda calcul , sont définis comme suit :
Les variables , constituées d'un caractère alphabétique minuscule, sont des L-termes.
Si x est une variable et si t est un L-terme une abstraction est le L-terme noté Lx.t.
Si t1 et t2 sont deux L-termes une application est le L-terme noté (t1 t2).
Un alias est un identificateur pour une abstraction .
Cet identificateur est composé d'au moins deux caractères alphabétiques minuscules à l'exclusion de tout autre caractère (pour le distinguer des variables) soit composé uniquement de chiffres décimaux auquel cas il dénote un entier de Church et est codé comme tel en interne.
Les seuls caractères blancs autorisés sont ceux qui séparent les deux L-termes d'une application.

```
L-terme        => variable | abstraction |application
abstraction    => 'L' '.' variable '.' L-terme | alias
alias          => identificateur | church
application    => '(' L-terme blancs L-terme ')'
blancs         => ' ' sblancs
sblancs        => ' ' sblancs | vide
identificateur => alpha alpha sident
sident         => alpha sident | vide
variable       => alpha
church         => chiffre shurch
schurch        => chiffre schurch | vide
chiffre        => '0' | '1' |'2'| ... | '9'
alpha          => 'a' | 'b' | 'c'| ... | 'z'
```

Exemples de L-termes

```
L'abstraction qui définit l'addition de deux entiers  :
Ln.Lm.Lf.Lx.((n f) ((m f) x))
L'abstraction qui définit cons  sur les listes :
Lx.Ly.((pair false) ((pair x) y)
L'addition de deux entiers : ((add 2) 4)
Une liste ayant 1 comme seul élément :
((cons 1) nil)
```

3 Les alias

On a vu lors de la définition de la grammaire des L-termes que les alias identifiaient des abstractions. Sans eux la moindre écriture d'un L-terme devient illisible .
Les alias sont soit prédéfinis et reconnus au lancement de l'interpréteur soit définis par l'utilisateur .
Dans le premier cas ils sont lus à partir d'un fichier texte incorporé au logiciel et chargés en mémoire.Ils constituent le prélude .
Dans le second cas l'utilisateur les définit dans la fenètre de commande. Ils sont alors soumis à l'analyseur syntaxique , convertis en un L-termes sans alias puis chargés en mémoire si l'analyse est concluante et enfin affichés dans la fenètre d'historique.
Dans le cas contraire un message d'erreur approprié est émis dans la fenètre des erreurs.
L'ensemble des alias définis par l'utilisateur et le prélude constituent l'environnement.

Les alias prédéfinis :
Ce sont :

Les entiers de Church
Les alias prédéfinis par la commande let.

Les entiers de Church
Les entiers de Church sont générés par une méthode spécifique et sont dénotés par des chiffres décimaux 0, 1, 2, .. 10, 11, ...comme il a été vu dans la grammaire des L-termes.

Par exemple 3 est un alias qui dénote Lf.Lx.(f (f (f x)))

Syntaxe de la commande let

let alias = abstraction
Notons ,en référence à la grammaire , que l'abstraction qui est mentionnée dans cette définition peut contenir des alias pourvu qu'ils fassent partie de l'environnement.

Liste des alias prédéfinis

```
La fonction identite
let id           = Lx.x

Les fonctions arithmétiques sur les entiers de Church
let succ         = Ln.Lf.Lx.((n f) (f x))
let add          = Ln.Lm.Lf.Lx.((n f) ((m f) x))

let mult         = Ln.Lm.Lf.Lx.((n (m f)) x)
let exp          = Ln.Lm.Lf.Lx.(((n m) f) x)

Les booléens

let true         =    Lu.Lv.u
let false        =    Lu.Lv.v

Les paires  . first , second

let pair        =    Lx.Ly.Lf.((f x) y)

let fst         =    Lp.(p Lx.Ly.x)
let snd         =    Lp.(p Lx.Ly.y)

Les opérations sur les listes

  let nil       =   Lx.x
  let null      =   Lp.(p Lx.Ly.x)
  let head      =   Lp.(fst (snd p))

  let tail      =   Lp.(snd (snd p))
  let cons      =   Lx.Ly.((pair false) ((pair x) y))


Les fonctions booléennes

  let and        =      Lp.Lq.(((if p) q) false)
  let or         =      Lp.Lq.(((if p) true) q)

  let not        =      Lp.(((if p) false) true)
```

```
  let iszero     =      Ln.((n Lx.false) true)


(((if cond) t1) t2)

  let if         =      Lx.Ly.Lz.((x y) z)

La fonction prédecesseur d'un entier

  let prfn       =      Lf.Lp.((pair false) (((fst p) (snd p)) (f (snd p))))
  let pred       =      Ln.Lf.Lx.(snd ((n (prfn f)) ((pair true) x)))



Les opérateurs de point fixe


let curry        =     Lf.(Lx.(f (x x)) Lx.(f (x x)))
let turing       =     (Lx.Lf.(f ((x x) f)) Lx.Lf.(f ((x x) f)))


Les fonctions récursives  définies par un point fixe
factorielle


 let pfact        =      Lf.Ln.(((iszero n) Lf.Lx.(f x)) ((mult n) (f (pred n))))
 let fact         =      (pfact (turing pfact))

la soustraction positive ((minus a) b)  : if (a > b ) then a-b else 0


  let pmin        =     Lf.Lm.Ln.(((iszero m) n) ((f (pred m)) (pred n)))
  let minus       =     (pmin (turing pmin))


Les operateurs relationnels sur les entiers

 ((gt p) p)   ;  if( p>q ) then true else false

  let notzero    = Ln.(not (iszero n))
  let gt         = Lm.Ln.(((notzero ((minus m) n)) true) false)

 la division entiere


 let pdiv    = Lf.Ln.Lm.Lp.((((gt ((mult p) m)) n) 0) (succ (((f n) m) (succ p))))
  let rdiv   = (pdiv (turing pdiv))
  let div    = Ln.Lm.(((rdiv n) m) 1)
```

4 Prise en main

L'exécutable vient d'être lançé et dans la fenêtre de commandes s'affiche :
Lambda Interpreter

Version 1.0

Commençons par tester les fonctions de base sur les entiers.
(Il sera bon d'avoir une fenêtre ouverte sur la page d'aide qui présente les alias standarts .)

```
(id 3)       donne Lf.Lx.(f (f (f x)))
((add 2) 3)  donne Lf.Lx.(f (f (f (f (f x)))))
(succ 2)     donne  Lf.Lx.(f (f (f x)))
(pred 4)     donne  Lf.Lx.(f (f (f x)))
((mult 2) 3) donne  Lf.Lx.(f (f (f (f (f (f x))))))
(fact 3)     donne  Lf.Lx.(f (f (f (f (f (f x))))))
Faites le calcul de (fact 5) . Vous devriez mettre un temps de calcul
relativement long.

((minus 7) 4)     donne  Lf.Lx.(f (f (f x)))
((minus 7) 7)     donne  Lf.Lx.x ( 0 )
((minus 3) 7)     donne  Lf.Lx.x ( 0 )
((div 7) 3)       donne  Lf.Lx.(f (f x))
(((if true) 5) 6) donne Lf.Lx.(f (f (f (f (f x))))) ( 5 )
((if false) 5) 6) donne Lf.Lx.(f (f (f (f (f (f x)))))) ( 6 )
(((if ((gt 7) 5)) 1) 0)  donne Lf.Lx.(f x) ( 1 )
(((if ((gt 5) 7)) 1) 0)  donne Lf.Lx.x ( 0 )

Créons maintenant quelques alias :

let double = Ln.((add n) n)
Dans la fenêtre d'historique s'affiche let double = Ln.((add n) n)
L'exécution de (double 3) donne  Lf.Lx.(f (f (f (f (f (f x))))))
La composition de deux fonctions n -> (f (g n)) s'écrit :
let comp = Lf.Lg.Ln.(f (g n)) qui s'affiche dans l'historique .
Composons double et succ .Par exemple  (double (succ 2)) -> 6
let dblsuc = ((comp double) succ)
(dblsucc 2) donne (double (succ 2)) =  Lf.Lx.(f (f (f (f (f (f x))))))


Une petite incursion vers la récursivité.
On définit l'addition add n m par:
add n 0 = n , if m=0
add n m = add (succ n) (pred m) , otherwise
Dans un premier temps on définit pradd comme l'abstraction portant sur une
fonction inconnue f qui devra vérifier les propriétés de add définies
ci dessus . Ce qui donne :
let pradd = Lf.Ln.Lm.(((if (iszero m)) n) ((f (succ n)) (pred m)))
Cette définition s'affiche dans l'historique.
La fonction f cherchée est (turing pradd) qui est le point fixe de pradd
(pradd (turing pradd)) = (turing pradd) ce qui nous amène à écrire :
let radd = (pradd (turing pradd))
On vérifie que :
((radd 3) 2) donne  Lf.Lx.(f (f (f (f (f x)))))

Quelques exercices sur les listes :
let lun = ((cons 1) nil)   . lun dénote la liste 1:[]
let ldeux = ((cons 2) lun) . ldeux dénote la liste 2:1:[]
```

```
let ltrois = ((cons 3) ldeux) . ltrois dénote la liste 3:2:1:[]
Pour implémanter append qui fusionne deux listes on écrira :
append z w = w , if (null z)
append z w = (cons (head z) (append (tail z) w)), otherwise
La mise en oeuvre se fera par la technique du point fixe vue pour radd
let papend = Lf.Lp.Lq.(((if (null p)) q) ((cons (head p)) ((f (tail p)) q)))
let append =(papend (turing papend))
On vérifiera que ((append ldeux) ltrois)  vaut bien ldeux:ltrois .
( utiliser head et tail ).
```