Práctica 3

314032324 Galeana Araujo, Emiliano
314011163 Miranda Sánchez, Kevin Ricardo

Facultad de Ciencias, UNAM

Fecha de entrega: Miercoles 3 de Octubre 2018

1 Descripción del programa

Expresiones del Calculo Lambda, el cálculo lambda consiste siplemente en tres términos y todas las combinaciones recursivas válidas de estos términos. Las definimos como sigue.

```
-- | Identifier. Tipo que define un nombre de variables como
-- |             una cadena de texto.
type Identifier = String

-- | Expr. Tipo que representa una expresion lambda sin tipos.
data Expr = Var Identifier
          | Lam Identifier Expr
          | App Expr Expr deriving(Eq)

-- | Substitution. Tipo que representa la sustitucion.
type Substitution = ( Identifier , Expr )
```

Por ejemplo, la siguiente expresión $\lambda x.\lambda y.xy$, la representaremos en Haskell como sigue. $\backslash x \to \backslash y \to (xy)$. Todo esto representa las expresiones del calculo lambda sin tipos y el reducto para la funcion de sustitucion.

Se realizaron las siguiente funciones que representan la semántica operacional en en calculo lambda sin tipos.

```
-- | frVars. Obtiene el conjunto de variables libres de una expresion.
frVars :: Expr → [Identifier]

-- | lkVars. Obtiene el conjunto de variables ligadas de una expresion.
lkVars :: Expr → [Identifier]

-- | incrVar. Dado un identificador, si este no termina en numero
-- |          le agrega el sufijo 1, en caso contrario toma el valor del numero
-- |          y lo incrementa en 1.
```

```
incrVar :: Identifier → Identifier


-- | alphaExpr. Toma una expresion lambda y devuelve una alpha-equivalente
-- |            utilizando la funcion incrVar hasta encontrar un nombre que no
-- |            aparezca en el cuerpo.
alphaExpr :: Expr → Expr

-- | subst. Aplica la sustitucion a la expresion dada.
subst :: Expr → Substitution → Expr

-- | beta. Aplica un paso de la beta reduccion.
beta :: Expr → Expr

-- | locked. Determina si una expresion esta bloqueada, es decir, no se pueden
-- |         hacer mas reducciones.
locked :: Expr → Bool

-- | eval. Evalua una expresion lambda aplicando beta reducciones hasta quedar
-- |       bloqueada.
eval :: Expr → Expr
```

2 Entrada y ejecución

El programa es interpretado por GCHI de la siguiente forma

```
~:ghci Practica3.hs
```

En el programa puede probar algunos ejemplos de ejecucion, esciribiendo simplemente el nombre del ejemplo que se quiere ejecutar.

2.1 Calculo lambda

En el programa se encuentran las lineas de codigo.

```
---FRVARS-----------
ejemplo = frVars (App (Lam "x" (App ( Var "x" ) ( Var "y" ) ) )(Lam "z" ( Var "z" ) ) )
ejemplo2 = frVars (Lam "f" (App (App (Var "f")(Lam "x"(App(App(Var "f")(Var
"x"))(Var "x" ))))(Lam "x"(App(App(Var "f")(Var "x" ))( Var "x")))))
---LKVARS---------
ejemplo3 = lkVars (App (Lam "x" (App ( Var "x" ) ( Var "y" ))) (Lam "z" ( Var
"z" )))
ejemplo4 = lkVars (Lam "f"(App(App(Var "f" )(Lam "x"(App(App(Var "f")(Var "x" ))
(Var "x" ))))(Lam "x" (App(App(Var "f" )(Var "x"))(Var "x" )))))
----INCRVAR----
ejemplo5 = incrVar "elem"
ejemplo6 = incrVar "x97"
```

```
-----ALPHAEXPR-------------------------
ejemplo7 = alphaExpr (Lam "x" (Lam "y" (App (Var "x" ) (Var "y" ))))
ejemplo8 = alphaExpr (Lam "x" (Lam "x1" (App ( Var "x" ) ( Var "x1"))))

--------SUBST------
ejemplo9 = subst (Lam "x" (App ( Var "x" ) ( Var "y" ) ) ) ( "y" , Lam "z" ( Var
"z" ))
ejemplo10 = subst (Lam "x" ( Var "y" )) ( "y" , Var "x" )

-----BETA-----
ejemplo11 = beta (App (Lam "x" (App ( Var "x" ) ( Var "y" ))) (Lam "z" ( Var "z"
)))
ejemplo20 = beta (App (Lam "n" (Lam "s" (Lam "z" (App ( Var "s" ) (App (App (
Var "n" ) ( Var "s" ) ) ( Var "z" ) ) ) ) ) ) (Lam "s" (Lam "z" ( Var "z" ) ) ) )
-----------LOCKED--------
ejemplo12 = locked (Lam "s" (Lam "z" ( Var "z" ) ) )
ejemplo13 = locked (Lam "x" (App (Lam "x" ( Var "x" ))( Var "z" )))

ejemplo14 = eval (App (Lam "n" (Lam "s" (Lam "z" (App ( Var "s" ) (App (App (
Var "n" )  ( Var "s" ) ) ( Var "z" ) ) ) ) ) ) (Lam "s" (Lam "z" ( Var "z" ) ) ) )
ejemplo15 = eval (App (Lam "n"(Lam "s"(Lam "z" (App (Var "s")(App (App (Var "n")
(Var "s"))(Var "z" ))))))(Lam "s" (Lam "z" (App ( Var "s" )(Var "z" )))))

cero = Lam "s" (Lam "z" (Var "z"))
uno = Lam "s1" (Lam "z1" (App (Var "s1") (Var "z1")))
suc = Lam "n" (Lam "s2" (Lam "z2" (App (Var "s2") (App (App (Var "n") (Var
"s2"))  (Var "z2")))))

ejemplo16 = eval (App suc cero)
ejemplo17 = eval (App suc uno)
```

Entonces, para ejecutar algun de los ejemplos, basta escribir el nombre de la siguiente manera:

```
*Practica3> ejemplo15
λs →λz → (s (s z))
*Practica3> ejemplo12
True
*Practica3> ejemplo3
["x","z"]
*Practica3> ejemplo9
λx → (x λz → z)
```

Nota, el regreso no está como en nuestra implementación, ya que tiene el simbolo λ, pero se puede ver que ambas representaciones son equivalentes.

3 Conclusiones

La parte de `locked` fue la más complicada, ya que hicimos varias implementaciones y con los ejemplos del PDF estaban bien, pero cuendo metíamos otros

ejemplos, no salía lo que esperábamos, y en algún momento pensamos en usar `locked` para hacer `eval`, al final no lo necesitamos y pudimos arreglar `locked` en su mayoría.

References

[1] Archivero, curso de Lenguajes de Programación 2019-1