Práctica 3

Integrando un núcleo funcional a `EAB`.

Favio E. Miranda Perea (favio@ciencias.unam.mx)
Pablo G. González López (pablog@ciencias.unam.mx)

Jueves 19 de septiembre de 2019

Fecha de entrega: Jueves 26 de Septiembre de 2019 a las 23:59:59.

En esta práctica extenderemos nuestro lenguaje de `EAB` con la implementación del cálculo lambda para convertirlo en un pequeño lenguaje de programación funcional al que llamaremos `MiniHs`.

1 Sintaxis

Primero agregaremos a las expresiones los constructores de una abstracción lambda (que de ahora en adelante llamaremos **función**) y el de la aplicación:

```
data Expr = ...
          | Fn Identifier Expr
          | App Expr Expr
```

Como ya hemos visto, existen varias opciones para representar la sintaxis de las funciones, sin embargo nosotros seguiremos utilizando una notación que las muestre en sintaxis abstracta de orden superior.

Ejemplo:

- **Notación Lógica:** $\lambda x.\lambda y.xy$
- **Nuestra notación:** `fn(x.fn(y.app(x, y)))`

De modo que agregaremos lo siguiente en la instancia de la clase `Show` para `Expr`:

```
instance Show Expr where
    show e = case e of
               (V x) -> "V[" ++ x ++ "]"
               ...
               (Fn x e) -> "fn(" ++ x ++ "." ++ (show e) ++ ")"
               (App e1 e2) -> "app(" ++ (show e1) ++ ", " ++ (show e2) ++ ")"
```

Extiende o implementa las siguientes funciones en el módulo `Sintax`:

1. (1 punto) `frVars`. Obtiene el conjunto de variables libres de una expresión.

```
frVars :: Expr -> [Identifier]
```

Ejemplo:

```
*Main> frVars (App (Fn "x" (App (V "x") (V "y"))) (Fn "z" (V "z")))
["y"]
*Main> frVars (Fn "f" (App (App (V "f") (Fn "x" (App (App (V "f") (V "x"
[]
```

2. (1 punto) `incrVar`. Dado un identificador, si este no termina en número le agrega el sufijo 1, en caso contrario toma el valor del número y lo incrementa en 1.

```
incrVar :: Identifier -> Identifier
```

Ejemplo:

```
*Main> incrVar "elem"
"elem1"
*Main> incrVar "x97"
"x98"
```

3. (1 punto) `alphaExpr`. Toma una expresión que involucre el ligado de una variable y devuelve una α-equivalente utilizando la función `incrVar` hasta encontrar un nombre que no aparezca en el cuerpo.

```
alphaExpr :: Expr -> Expr
```

Ejemplo:

```
*Main> alphaExpr (Fn "x" (Fn "y" (App (V "x") (V "y"))))
fn(x1.fn(y.app(V[x1], V[y])))
*Main> alphaExpr (Fn "x" (Fn "x1" (App (V "x") (V "x1"))))
fn(x1.fn(x2.app(V[x1], V[x2])))
```

4. (2 puntos) `subst`. Aplica la sustitución a la expresión dada. (Utiliza la función `alphaExpr` para implementar una función total)

```
subst :: Expr -> Substitution -> Expr
```

Ejemplo:

```
*Main> subst (Fn "x" (App (V "x") (V "y"))) ("y", Fn "z" (V "z"))
fn(x.app(V[x], fn(z.V[z])))
*Main> subst (Fn "x" (V "y")) ("y", V "x")
fn(x1.V[x])
```

2 β-reducción

La beta reducción es simplemente un paso de sustitución, que reemplaza la variable ligada por una función por el argumento de la aplicación.

$app(fn(x.a), y) \rightarrow^{\beta} a[x := y]$

3 Evaluación

La regla anterior es no determinista, por lo que no podemos implementar directamente la β-reducción para evaluar nuestras expresiones. Antes debemos analizar esta regla para encontrar cómo realizar la evaluación de manera estratégica.

Buscamos reducir una expresión del cálculo lambda hasta encontrar una expresión e que no se pueda reducir más. Es decir, no exista una e' tal que $e \rightarrow^{\beta} e'$. A esta expresión e se le conoce como **forma normal**. El predicado para decidir si una expresión está en forma normal se define del siguiente modo:

$$\frac{}{x\ normal}\ F.N.(Variable)$$

$$\frac{e\ normal}{\lambda x.e\ normal}\ F.N.(Abstraccion\ lambda)$$

$$\frac{e_1\ normal \quad e_2\ normal \quad e_1 \neq \lambda\alpha.\eta}{(e_1 e_2)\ normal}\ F.N.(Aplicacion)$$

Con esto en mente definiremos las reglas de nuestra estrategia de evaluación del mismo modo que lo hemos hecho hasta ahora, evaluando de izquierda a derecha de a un paso a la vez hasta que, eventualmente, alcancemos una expresión en forma normal.

$$\frac{e \rightarrow e'}{fn(x.e) \rightarrow fn(x.e')}$$

$$\frac{e_1 \rightarrow e_1'}{app(e_1, e_2) \rightarrow app(e_1', e_2)}$$

$$\frac{e_1\ normal \quad e_2 \rightarrow e_2'}{app(e_1, e_2) \rightarrow app(e_1, e_2')}$$

$$\frac{e_2\ normal}{app(fn(x.e), e_2) \rightarrow e[x := e_2]}$$

Implementa o extiende las siguientes funciones en el módulo `Semantic`:

1. (2,5 puntos) `eval1`. Devuelve la transición tal que `eval1 e = e'` syss $e \to e'$.

```
eval1 :: Expr -> Expr
```

   Ejemplo:

```
*Main> eval1 (App (Fn "x" (App (V "x") (V "y")))
    (Fn "z" (V "z")))
app(fn(z.V[z]), V[y])
*Main> eval1 (Lt ((App (Fn "x" (Add (V "x") (I
    20)))) (I 10))) (I 20))
lt(add(N[10], N[20]), N[20])
```

2. (0,25 puntos) `evals`. Devuelve la transición tal que `evals e = e'` syss $e \to^* e'$ y e' está bloqueado.

```
evals :: Expr -> Expr
```

   Ejemplo:

3. (0,25 puntos) `evale`. Devuelve la evaluación de un programa tal que `evale e = e'` syss $e \to^* e'$ y e' es un valor. En caso de que e' no esa un valor deberá mostrar un mensaje de error particular del operador que lo causó.

```
evale :: Expr -> Expr
```

   Ejemplo:

¡Suerte!