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: Miércoles 2 de Octubre 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 muestre las expresiones 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")) (V "x")))) (Fn "x"
    (App (App (V "f") (V "x")) (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 Semántica

2.1 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.

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

2.2 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 bloqueada o en forma normal.

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

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

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

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

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(I[10], I[20]), I[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:

   ```
   *Main> evals (App (Fn "x" (If (Or (V "x") (B
       False)) (Add (I 1) (I 1)) (Succ (Pred (I 0)))
       )) (B False))
   I[0]
   *Main> evals (App (App (Fn "x" (If (Or (V "x") (B
        False)) (Add (I 1) (I 1)) (Succ (Pred (I 0))
       ))) (B False)) (V "x"))
   app(if(or(B[False], B[False]), add(I[1], I[1]),
       succ(pred(N[0]))), V[x])
   ```

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:

```
*Main> evale (Let "f" (Fn "x" (If (Or (V "x") (B
    False)) (Add (I 1) (I 1)) (Succ (Pred (I 0)))
    )) (App (V "f") (B False)))
I[0]
*Main> evale (Let "f" (Fn "x" (If (Or (V "x") (B
    False)) (Add (I 1) (I 1)) (Succ (Pred (I 0)))
    )) (App (App (I 1) (V "f")) (B False)))
*** Exception: [App] Expects a Function as first
    argument.
```

3 Funciones recursivas

En este punto, nuestro evaluador ya esta dotado de un poderoso núcleo funcional que nos permite definir nuestras propias funciones para realizar cualquier operación que deseemos. Sin embargo, aún falta un concepto esencial en cualquier lenguaje de programación funcional, la recursión.

Como se vio en clase, el cálculo lambda puro es capaz de expresar funciones recursivas sin la necesidad de utilizar ningún operador ajeno a este. Esto implica para definir las funciones recursivas simplemente tendremos que combinar de algún modo las expresiones que ya tenemos y no tendremos que programar nada adicional.

3.1 Combinadores de punto fijo

Un λ-término cerrado F es un combinador de punto fijo si y solamente si cumple alguna de las siguientes condiciones:

1. $Fg \rightarrow_\beta^* g(Fg)$
2. $Fg =_\beta g(Fg)$, es decir, existe un término L tal que $Fg \rightarrow_\beta^* L$ y $g(Fg) \rightarrow_\beta^* L$

En clase ya se han visto varios de los combinadores de punto fijo que se han descubierto, por lo que no tocaremos más este tema.

3.2 Implementación

Empezaremos con añadir a nuestro lenguaje una expresión que se utilizará para definir funciones recursivas la cuál llamaremos `recurFn`:

```
e ::= x | n | true | false
    ...
    | recurFn f x => e
```

Ahora hay que añadirle una semántica.

Como ya dijimos que esta nueva expresión es simple azúcar sintáctica, no hay que tocar los módulos `Sintax` y `Semantic`. En su lugar tendremos que

tomar la expresión que nos devuelve el analizador sintáctico y transformarla a una expresión del analizador semántico. Esto se hace específicamente en la función `parserExprToExpr` del `Main`.

Descarga el fichero `BAE.zip` que contiene la nueva versión del analizador sintáctico.

Realiza lo siguiente en el módulo `Main`:

1. (1 punto) `fix`. Define una constante llamada `fix` que implemente algún combinador de punto fijo[1].

```
fix = ...
```

2. (1 punto) `parserExprToExpr`. Agrega la definición para transformar la expresión de la función recursiva del analizador sintáctico a su equivalente del analizador semántico.

```
...
parserExprToExpr (Parser.RecurFn f x e) = ...
```

4 Punto Extra: Funciones de varios argumentos

4.1 *Curry-ficación*

Curry-ficar es el proceso de transformar una función que toma varios argumentos en una función que tome solo un argumento y regrese otra función que acepte los demás argumentos, de uno en uno, que la función original recibiría.

Por ejemplo:

```
f :: a -> (b -> c)
```

es la versión *curry-ficada* de

```
g :: (a, b) -> c
```

es decir,

```
f = curry g
g = uncurry f
```

4.2 Implementación

Sustituiremos en nuestro lenguaje las expresiones de funciones y funciones recursivas por:

```
e ::= x | n | true | false
    ...
    | fn [x] => e
    | recurFn f [x] => e
```

[1] Revisa la *Nota de clase 6: Introducción al Paradigma Funcional, Cálculo Lambda sin Tipos.*

Cada `[x]` es una lista de identificadores separados por espacio.

Para realizar la implementación de esta sección de la práctica deberás descargar el fichero `BAE[MultipleArgs].zip`, el cuál tiene la implementación del analizador sintáctico para este tipo de funciones.

Los módulos `Sintax` y `Semantic` no sufrirán ningún cambio, pues las funciones de varios argumentos son también azúcar sintáctica.

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

1. `currying`. Toma una función de varios argumentos del analizador sintáctico y aplica la *curry-ficación* para transformarla en funciones de un solo argumento.

   ```
   currying :: Parser.Expr -> Sintax.Expr
   ```

   (**Nota:** Esta función estará definida únicamente para el constructor `Fn Identifiers Expr`)

2. `parserExprToExpr`. Agrega la definición para transformar las expresiones de función y función recursiva del analizador sintáctico a su equivalente *curry-ficada* del analizador semántico.

   ```
   ...
   parserExprToExpr (Parser.Fn args e) = ...
   parserExprToExpr (Parser.RecurFn f args e) = ...
   ```

5 Formato de entrega

Deberás compilar el evaluador ejecutando `make` desde el directorio raíz. Se generará un archivo ejecutable llamado `BAEi`. Ejecuta `make run`. Si todo funciona como debería obtendrás una salida como la siguiente:

```
$ make run

mkdir -p build
ghc src/Main.hs -isrc -outputdir build/ -o BAEi
./BAEi ./demo/goodExample.bae
Program:
let(mul(add(N[1], N[2]), succ(pred(N[10]))), x1.let(
    eq(N[1], N[1]), y.if(or(and(B[True], gt(V[x1], N
    [-4])), B[False]), or(not(V[y]), lt(V[x1], N[31]))
    , lt(V[x1], N[-4])))) : Boolean
Evaluation:
B[True]
```

En el directorio `demo` hay un par de programas para verificar que se ejecutan de manera correcta. Para ejecutarlos basta llamar al evaluador `BAEi` seguido de la ruta del programa. Ejemplo:

```
./BAEi ./demo/factorial.bae
```

La estructura de este entregable será parecida a la siguiente dependiendo de cuántas especificaciones se implementen:

```
<Nombre><Apellido>
    |-- Practica3
        |-- readme.pdf
        |-- src
            |-- BAE (Especificacion Obligatoria)
            |   |-- Makefile
            |   |-- ...
            |-- BAE[MultipleArgs] (Especificacion
                Voluntaria)
                |-- Makefile
                |-- ...
```

Atención

Será un **requisito de entrega** integrar correctamente los módulos `Sintax` y `Semantic` al analizador sintáctico.

¡Suerte!