Práctica Extra

Implementación de la representación anónima de términos λ
(Índices de de-Bruijn)

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

Lunes 26 de noviembre de 2018

Fecha de entrega: Domingo 9 de diciembre de 2018 a las 23:59:59.

Reglas:

Los lineamientos de entrega serán los mismos de las prácticas anteriores salvo los siguientes:

1. La calificación de esta práctica sustituirá a la menor calificación del laboratorio, siempre y cuando sea **aprobatoria**.
2. El desarrollo y la entrega deberá ser de manera **individual**.
3. La fecha de entrega es definitiva. **No habrá retardos ni prorrogas**.

Anteriormente representamos las expresiones del cálculo lambda con nombres en las variables con la siguiente gramática:

$$e ::= x|ee|\lambda x.e$$

Ahora la idea para representar estas mismas expresiones será:

Dada una expresión lambda, representar una variable ligada apuntando directamente al símbolo lambda que la liga en el árbol de sintaxis abstracta correspondiente, es decir mediante el número de lambdas que es necesario "saltar" hasta encontrar la lambda que liga a la variable en cuestión.

Estos números son conocidos como los índices de de-Bruijn:

$$e ::= n|ee|\lambda.e$$

Lee la nota adicional correspondiente a este tema.

1 Implementación

La representación de las expresiones del cálculo lambda con nombres será:

```
type Identifier = String

data ExprL = VL Identifier
           | LL Identifier ExprL
           | AL ExprL ExprL
```

Y la de las expresiones con índices de de-Bruijn:

```
type Index = Int

data ExprB = IB Index
           | LB ExprB
           | AB ExprB ExprB
```

Además que la sustitución de representará como:

```
type Substitution = (Index, ExprB)
```

Implementa las siguientes funciones:

1. (1 punto) `ctx`. Obtiene el contexto canónico de una expresión.

   Recuerda que el contexto canónico es una sucesión de variables libres de una expresión donde si existían repeticiones de variables en el árbol sintáctico, se mantiene únicamente la última ocurrencia de dicha variable. Para facilitar la obtención de los índices lo representaremos de manera inversa a como se describe en la nota, es decir el primer elemento corresponderá al índice 0, el segundo al 1 y así sucesivamente.

   ```
   ctx :: ExprL -> [Identifier]
   ```

   Ejemplo:

   ```
   *Main> ctx (LL "x" (LL "y" (AL (VL "u") (AL (VL "x") (AL (VL "y")
   (AL (VL "z") (AL (VL "z") (AL (VL "y") (VL "v")))))))))
   ["v", "z", "u"]
   *Main> ctx (AL (VL "u") (AL (VL "v") (AL (VL "x") (AL (VL "y")
   (AL (VL "z") (AL (VL "x") (AL (VL "x") (VL "v")))))))))
   ["v", "x", "z", "y", "u"]
   ```

2. (2.5 puntos) `qn`. Dado un contexto de índices y una expresión lambda obtiene su representación anónima.

   ```
   qn :: ([Identifier], ExprL) -> ExprB
   ```

   Ejemplo:

   ```
   *Main> qn (["x"], LL "z" (AL (VL "z") (AL (VL "x") (LL "y" (AL (VL "z")
   (AL (VL "x") (VL "y")))))))
   LB (AB (IB 0) (AB (IB 1) (LB (AB (IB 1) (AB (IB 2) (IB 0))))))
   -- qn (["x"], \z ->(z (x \y ->(z (x y)))))
   ```

```
-- (\.(0 (1 (\.(1 (2 0))))))
*Main> qn (["y", "x"], LL "z" (AL (VL "z") (AL (VL "x") (VL "y"))))
LB (AB (IB 0) (AB (IB 2) (IB 1)))
-- qn (["y, x"], \z ->(z (x y)))
-- (\.(0 (2 1)))
```

3. (0.5 puntos) `newVar`. Dado un contexto de nombres obtiene una nueva variable y la agrega al contexto dado.

   Hint: Para encontrar esta nueva variable usa la función `incrVar` implementada anteriormente.

   ```
   newVar :: [Identifier] -> [Identifier]
   ```

   Ejemplo:

   ```
   *Main> newVar ["y", "z"]
   ["x", "y", "z"]
   *Main> newVar ["x", "x2", "x1"]
   ["x3", "x", "x2", "x1"]
   ```

4. (2.5 puntos) `pn`. Dado un contexto de nombres y una expresión anónima devuelve su representación correspondiente en el cálculo lambda con nombres.

   ```
   pn :: ([Identifier], ExprB) -> ExprL
   ```

   Ejemplo:

   ```
   *Main> pn (["x"], LB (AB (IB 0) (AB (IB 1) (LB (AB (IB 1) (AB (IB 2)
   (IB 0)))))))
   LL "x1" (AL (VL "x1") (AL (VL "x") (LL "x2" (AL
   (VL "x1") (AL (VL "x") (VL "x2"))))))
   -- pn (["x"], (\.(0 (1 (\.(1 (2 0)))))))
   -- \x1 ->(x1 (x \x2 ->(x1 (x x2))))
   *Main> pn (["y", "x"], LB (AB (IB 0) (AB (IB 2) (IB 1))))
   LL "x1" (AL (VL "x1") (AL (VL "x") (VL "y")))
   -- pn (["y", "x"], \.(0 (2 1)))
   -- \x1 ->(x1 (x y))
   ```

5. (1 punto) `shift`. Desplaza los índices de una expresión anónima dado un parámetro de corte.

   ```
   shift :: (Int, Int, ExprB) -> ExprB
   ```

   Ejemplo:

   ```
   *Main> shift (1, 0, LB (AB (IB 0) (IB 2)))
   LB (AB (IB 0) (IB 3))
   -- shift (1, 0, (\.(0 2)))
   ```

```
-- (\.(0 3))
*Main> shift (1, 2, LB (AB (IB 0) (AB (IB 2) (IB 1))))
LB (AB (IB 0) (AB (IB 2) (IB 1)))
-- shift (1, 2, (\.(0 (2 1))))
-- (\.(0 (2 1)))
```

6. (1 punto) `subst`. Aplica la sustitución a la expresión anónima.

```
subst :: ExprB -> Substitution -> ExprB
```

Ejemplo:

```
*Main> subst (LB (AB (IB 0) (AB (IB 2) (IB 1)))) (1, LB (AB (IB 0)
(IB 2)))
LB (AB (IB 0) (AB (LB (AB (IB 0) (IB 3))) (IB 1)))
-- subst ((\.(0 (2 1)))) (1 := (\.(0 2)))
-- (\.(0 ((\.(0 3)) 1)))
```

7. (0.5 puntos) `eval1`. Aplica un paso de la reducción de una expresión anónima.

   Las reglas de evaluación son las siguientes:

$$\frac{t \to t'}{\lambda t \to \lambda t'}\ Lam$$

$$\frac{t_1 \to t_1'}{t_1 t_2 \to t_1' t_2}\ App$$

$$\frac{}{(\lambda t)s \to^{\beta} shift(-1, 0, t[0 := shift(1, 0, s)])}\ Beta$$

```
eval1 :: ExprB -> ExprB
```

Ejemplo:

```
*Main> eval1 (AB (LB (AB (LB (IB 1)) (IB 0))) (LB (AB (IB 2) (AB (IB 1)
(IB 0)))))
AB (LB (LB (AB (IB 3) (AB (IB 2) (IB 0))))) (LB (AB (IB 2) (AB (IB 1)
(IB 0))))
-- eval1 (((\.((\.1) 0)) (\.(2 (1 0)))))
-- ((\.(\.(3 (2 0)))) (\.(2 (1 0))))
```

8. (0.5 puntos) `locked`. Determina si una expresión anónima esta bloqueada, es decir, no se pueden hacer más reducciones.

```
locked :: ExprB -> Bool
```

Ejemplo:

```
*Main> locked (IB 1)
True
*Main> locked (LB (AB (LB (IB 0)) (LB (IB 0))))
False
```

9. (0.5 puntos) `eval`. Evalúa una expresión anónima hasta quedar bloqueada.

```
eval :: ExprB -> ExprB
```

10. (Ejercicio moral) Verifica que tu programa cumple con la relación entre `pn` y `qn`. Esto es:

`alphaExpr(e, pn(Γ, qn(Γ, e))) == True` y `e == qn(Δ, pn(Δ, e))`

¡Suerte!