Jose Revete
19-10040

Tarea 1 (CI-3661)

1. **Considere la estructura de datos Conjunto, que representa conjuntos potencialmente infinitos. Tomando en cuenta la definición anterior, debe implementar entonces cada una de las siguientes funciones (válidas independientemente del tipo concreto que tome a y sin cambiar sus firmas):**

R:

```haskell
type Conjunto a = a -> Bool

-- a)
miembro :: Conjunto a -> a -> Bool
miembro c x = c x

-- b)
vacio :: Conjunto a
vacio = \_ -> False

-- c)
singleton :: (Eq a) => a -> Conjunto a
singleton x = \y -> x == y

-- d)
desdeLista :: (Eq a) => [a] -> Conjunto a
desdeLista xs = \x -> elem x xs

-- e)
complemento :: Conjunto a -> Conjunto a
complemento c = \x -> not (c x)

-- f)
union :: Conjunto a -> Conjunto a -> Conjunto a
union c1 c2 = \x -> c1 x || c2 x

-- g)
interseccion :: Conjunto a -> Conjunto a -> Conjunto a
interseccion c1 c2 = \x -> c1 x && c2 x

-- h)
diferencia :: Conjunto a -> Conjunto a -> Conjunto a
diferencia c1 c2 = \x -> c1 x && not (c2 x)

-- i)
transformar :: (b -> a) -> Conjunto a -> Conjunto b
transformar f c = \x -> c (f x)
```

2. Considere el tipo de datos ArbolMB:

```haskell
data ArbolMB a = Vacio
               | RamaM a (ArbolMB a)
               | RamaB a (ArbolMB a) (ArbolMB a)
    deriving Show

{-
a)
Vacio :: ArbolMB a
RamaM :: a -> ArbolMB a -> ArbolMB a
RamaB :: a -> ArbolMB a -> ArbolMB a -> ArbolMB a
-}

{-
b)
transformarVacio :: b -> b
transformarRamaM :: a -> b -> b
transformarRamaB :: a -> b -> b -> b
 -}

-- c)
plegarArbolMB :: b
              -> (a -> b -> b)
              -> (a -> b -> b -> b)
              -> ArbolMB a
              -> b
plegarArbolMB transVacio transRamaM transRamaB = plegar
    where
        plegar Vacio = transVacio
        plegar (RamaM x y) = transRamaM x (plegar y)
        plegar (RamaB x y z) = transRamaB x (plegar y) (plegar z)


-- d)
sumarArbolMB :: (Num a) => ArbolMB a -> a
sumarArbolMB = plegarArbolMB transVacio transRamaM transRamaB
    where
        transVacio = 0
        transRamaM x acc = x + acc
        transRamaB x acc1 acc2 = x + acc1 + acc2


-- e)
aplanarArbolMB :: ArbolMB a -> [a]
aplanarArbolMB = plegarArbolMB transVacio transRamaM transRamaB
    where
        transVacio = []
        transRamaM x acc = x : acc
        transRamaB x acc1 acc2 = acc1 ++ [x] ++ acc2
```

```
-- f)
analizarArbolMB :: (Ord a) => ArbolMB a -> Maybe (a, a, Bool)
analizarArbolMB = plegarArbolMB transVacio transRamaM transRamaB
    where
        transVacio = Nothing
        transRamaM x Nothing = Just (x, x, True)
        transRamaM x (Just (min, max, b)) = Just (min, max, b && min < x && x <
max)
        transRamaB x Nothing Nothing = Just (x, x, True)
        transRamaB x (Just (min1, max1, b1)) Nothing = Just (min1, max1, b1 &&
min1 < x && x < max1)
        transRamaB x Nothing (Just (min2, max2, b2)) = Just (min2, max2, b2 &&
min2 < x && x < max2)
        transRamaB x (Just (min1, max1, b1)) (Just (min2, max2, b2)) = Just
(min1, max2, b1 && b2 && min1 < x && x < max2)
```

{-

g) Para crear una función plegarGen para un tipo de datos Gen a con n constructores diferentes, la función plegarGen deberá tomar n-funciones como argumento (además del valor de tipo Gen a que se desea plegar), dado que cada constructor es diferente y puede requerir un tratamiento distinto.

h) La función predefinida en el Preludio de Haskell que tiene una firma y un comportamiento similar a la función de plegado propuesta es 'foldr'. Esta función tiene un tipo de la forma: `foldr :: (a -> b -> b) -> b -> [a] -> b`. Aplica de manera recursiva una función a cada elemento de una lista, acumulando un resultado.

-}

3. **Se desea implementar entonces un monad que represente cómputos secuenciales:**

R:

```
newtype Secuencial s a = Secuencial (s -> (a, s))

-- c)
return :: a -> Secuencial s a
return x = Secuencial $ \estado -> (x, estado)

-- d)
instance Monad (Secuencial s) where
     return :: a -> Secuencial s a
     return x = Secuencial $ \estado -> (x, estado)
```

```
(>>=) :: Secuencial s a -> (a -> Secuencial s b) -> Secuencial s b
(Secuencial programa) >>= transformador =
    Secuencial $ \estadoInicial ->
        let (resultado, nuevoEstado) = programa estadoInicial
            (Secuencial nuevoPrograma) = transformador resultado
        in nuevoPrograma nuevoEstado
```

{-

a) Dado que en Haskell, las instancias para (por lo menos este caso) Monads, deben ser de la forma <*Constructor*> <*Argumento*>, donde Constructor es el nombre del tipo de dato y Argumento es el argumento de tipo que recibe el constructor.

b)

```
return :: a -> Secuencial s a
   >>= :: Secuencial s a -> (a -> Secuencial s b) -> Secuencial s b
   >> :: Secuencial s a -> Secuencial s b -> Secuencial s b
   fail :: String -> Secuencial s a
```

e)

- Identidad izquierda: *return a >>= h == h a*

Demostración:

return a >>= h
= Secuencial $ \estadoInicial -> (a, estadoInicial) >>= h
= Secuencial $ \estadoInicial -> h a estadoInicial
= h a

- Identidad derecha: *m >>= return == m*

Demostración:

m >>= return
= Secuencial $ \estadoInicial -> m estadoInicial >>= return
= Secuencial $ \estadoInicial -> m estadoInicial
= m

- Asociatividad: *(m >>= g) >>= h == m >>= (\x -> g x >>= h)*

Demostración:

(m >>= g) >>= h
= Secuencial $ \estadoInicial -> (m >>= g) estadoInicial >>= h
= Secuencial $ \estadoInicial -> (Secuencial $ \estado -> (g (fst (m estado)), snd (m estado))) estadoInicial >>= h
= Secuencial $ \estadoInicial -> h (g (fst (m estadoInicial))) (snd (m estadoInicial))
= Secuencial $ \estadoInicial -> m estadoInicial >>= (\x -> g x >>= h)
= m >>= (\x -> g x >>= h)

-}

Investigación:

1. **La programación funcional está basada en el Lambda Cálculo, propuesto por Alonzo Church. Dicho cálculo está basado en llamadas λ–expresiones. Tomando esta definición en cuenta, conteste las siguientes preguntas:**

R:

{-

a) *(λx . λy . x y y) (λz . z O) L*

= eval((λx . λy . x y y) (λz . z O)) eval(L)

= (λy . x y y) [x := eval(λz . z O)] eval(L)

(Paréntesis para indicar que se sustituye x por *(λz . z O)*)

= [x := (λz . eval(z O))]

= [x := (λz . z eval(O))]

(Continuamos con la sustitución)

= (λy . x y y) [x := (λz . z eval(O))] eval(L)

= (λy . (λz . z eval(O)) y y) eval(L)

(Resolvemos *(λz . z eval(O)) y*)

= (λz . z eval(O)) y

= (z eval(O)) [z := eval(y)]

= (z eval(O)) [z := y]

= (y eval(O))

(Continuamos con la sustitución)

= (λy . y eval(O) y) eval(L)

= (y eval(O) y) [y:= eval(L)]

= (eval(L) eval(O) eval(L))

= L O L

b) Si, por ejemplo:

E= λx . x

F= (λy . y y) (λy . y y)

Si evaluamos primero E F:

= (λx . x) (λy . y y) (λy . y y)

= (x) [x := eval(λy . y y)] (λy . y y)

= (x) [x := (λy . y y)] (λy . y y)

= (λy . y y) (λy . y y)

= (y y) [y := eval(λy . y y)]

= (y y) [y := (λy . y y)]

= (λy . y y) (λy . y y)

Si evaluamos primero F E:

= (λy . y y) (λy . y y) (λx . x)

= *(y y) [y := eval(λy . y y)] (λx . x)*
= *(y y) [y := (λy . y y)] (λx . x)*
= *(λy . y y) (λx . x)*
= *(y y) [y := eval(λx . x)]*
= *(y y) [y := (λx . x)]*
= *(λx . x) (λx . x)*
= *x [x := eval(λx . x)]*
= *x [x := (λx . x)]*
= *(λx . x)*
Por lo tanto, el orden en el que se evalúan las expresiones es relevante.

c) Cuando se permiten identificadores repetidos, la evaluación de una expresión lambda como *(λx.E)F* debe ajustarse, dado que si al sustituir x por F, si F contiene una variable que ya está en uso en E, se debe renombrar dicha variable para evitar confusión. Si no se renombra, la evaluación podría no ser correcta, ya que las variables se mezclarán (al menos visualmente).
-}

2. **Considere:**

```
id :: a -> a
id x = x

const :: a -> b -> a
const x _ = x

subs :: (a -> b -> c) -> (a -> b) -> a -> c
subs x y z = x z (y z)
```

R:
{-
a) subs (id const) const id
= *(id const) id (const id)*
= *const id (const id)*
= *id*

b) subs id id (subs id id)
= *id (subs id id) (id (subs id id))*
= *(subs id id) (subs id id)*
= *subs id id (subs id id)*

c) *id' :: a -> a*
id' = subs (const (const x)) (const (const x)) ()

d) Las funciones propuestas (id, const, y subs) están directamente relacionadas con el cálculo de combinadores SKI, de la siguiente manera:

- **id es como el combinador I:** La función id devuelve el mismo valor que recibe, exactamente lo que hace el combinador I en SKI. Es la identidad.
- **const es como el combinador K:** La función const toma dos argumentos y siempre devuelve el primero, ignorando el segundo. Esto es exactamente lo que hace el combinador K.
- **subs es como el combinador S:** La función subs aplica una función a dos argumentos que, a su vez, dependen del mismo valor. Esto es lo mismo que hace el combinador S en SKI, que "distribuye" funciones.

-}