Universidad Simón Bolívar

Departamento de Computación y Tecnología de la Información

CI3661 – Laboratorio de Lenguajes de Programación

Trimestre: Septiembre - Diciembre 2023

Profesor: Ricardo Monascal

Estudiante: Junior Miguel Lara Torres, Carnet: 17-10303

Tarea 1: Programación Funcional y Haskell (15 pts)

IMPLEMENTACION

- Parte 1:
  - a) Recibimos un conjunto y un elemento. Así aplicamos directamente la "función" al elemento dado.

```haskell
miembro :: Conjunto a → a → Bool
-- miembro conjunto e = conjunto e
miembro conjunto = conjunto
```

  - b) Para todos los elementos del conjunto, cual sea, retornamos False.

```haskell
vacio :: Conjunto a
vacio _ = False
```

  - c) Como "Conjunto a = a -> Bool" entonces la firma realmente de singleton es "a -> a -> Bool", por tanto recibimos dos elementos e1 y e2 y verificamos si siempre e1 es igual a e2.

```haskell
singleton :: (Eq a) ⇒ a → Conjunto a
-- singleton e1 e2 = e1 == e2
singleton = (==)
```

- d) Como “Conjunto a = a -> Bool” entonces la firma realmente de desdeLista es “[a] -> a -> Bool”, por tanto, recibimos una lista y un elemento y verificamos que dicho elemento pertenezca o no a la lista.

```haskell
desdeLista :: (Eq a) ⇒ [a] → Conjunto a
desdeLista lista e = e `elem` lista
```

- e) Como “Conjunto a = a -> Bool” entonces la firma realmente de complemento es “Conjunto a -> a -> Bool”, por tanto, recibimos un conjunto y un elemento y negando el conjunto respectivo tendríamos el complemento.

```haskell
complemento :: Conjunto a → Conjunto a
complemento conjunto e = not (conjunto e)
```

- f) Como “Conjunto a = a -> Bool” entonces la firma realmente de unión es “Conjunto a -> Conjunto a -> a -> Bool”, por tanto, recibimos dos conjuntos y un elemento para que de esta forma realicemos la unión mediante “or” de dicho elemento sobre los conjuntos dados.

```haskell
union :: Conjunto a → Conjunto a → Conjunto a
union conjunto1 conjunto2 e = conjunto1 e || conjunto2 e
```

- g) Como “Conjunto a = a -> Bool” entonces la firma realmente de intersección es “Conjunto a -> Conjunto a -> a -> Bool”, por tanto, recibimos dos conjuntos y un elemento para que de esta forma realicemos la intersección mediante “and” de dicho elemento sobre los conjuntos dados.

```haskell
interseccion :: Conjunto a → Conjunto a → Conjunto a
interseccion conjunto1 conjunto2 e = conjunto1 e && conjunto2 e
```

- h) Como “Conjunto a = a -> Bool” entonces la firma realmente de diferencia es “Conjunto a -> Conjunto a -> a -> Bool”, por tanto, recibimos dos conjuntos y un elemento para que de esta forma realicemos el filtrado de los elementos del primer conjunto con los negados del segundo conjunto mediante un “and”.

```haskell
diferencia :: Conjunto a -> Conjunto a -> Conjunto a
diferencia conjunto1 conjunto2 e = conjunto1 e && not (conjunto2 e)
```

- i) Como “Conjunto a = a -> Bool” entonces la firma realmente de transformar es “(b->a) -> Conjunto a -> b -> Bool”, por tanto, recibimos la función de transformación que va de b hacia a, un conjunto y un elemento de tipo b para que de esta forma realicemos verificación de miembro sobre ese conjunto dado de la transformación del elemento dado.

```haskell
transformar :: (b -> a) -> Conjunto a -> Conjunto b
transformar t conjunto1 e = miembro conjunto1 $ t e
```

- Parte 2:

  - a)

```haskell
Vacio :: ArbolMB a

RamaM :: a → ArbolMB a → ArbolMB a

RamaB :: a → ArbolMB a → ArbolMB a → ArbolMB a
```

  - b)

```haskell
transformarVacio :: b

transformarRamaM :: a → b → b

transformarRamaB :: a → b → b → b
```

  - c)

```haskell
plegarArbolMB :: b
              → (a → b → b)
              → (a → b → b → b)
              → ArbolMB a
              → b
```

```haskell
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)

```haskell
sumarArbolMB :: (Num a) => ArbolMB a -> a

sumarArbolMB = plegarArbolMB transVacio transRamaM transRamaB
    where
        transVacio       = 0
        transRamaM x y   = x + y
        transRamaB x y z = (x + y) + z
```

- e)

```haskell
aplanarArbolMB :: ArbolMB a -> [a]

aplanarArbolMB = plegarArbolMB transVacio transRamaM transRamaB
    where
        transVacio       = []
        transRamaM x y   = [x] ++ y
        transRamaB x y z = y ++ [x] ++ z
```

- f)

```haskell
analizarArbolMB :: (Ord a) => ArbolMB a -> Maybe (a, a, Bool)
analizarArbolMB = plegarArbolMB transVacio transRamaM transRamaB
    where
        transVacio = Nothing
        transRamaM x y = case y of
                Just (minV, maxV, isOrd) -> Just (min x minV, max x maxV, isOrd && x <= maxV)
                Nothing -> Just (x, x, True)
        transRamaB x y z = case y of
                Just (minV, maxV, isOrd) -> case z of
                                        Just (minV', maxV', isOrd') -> do
                                            let newMin = min x $ min minV minV'
                                            let newMax = max x $ max maxV maxV'
                                            let newIsOrd = isOrd && isOrd' && minV <= x && x <= maxV'
                                            Just (newMin, newMax, newIsOrd)
                                        Nothing -> Just (min x minV, max x maxV, isOrd && minV <= x)
                Nothing -> case z of
                        Just (minV', maxV', isOrd') -> Just (min x minV', max x maxV', isOrd' && x <= maxV')
                        Nothing -> Just (x, x, True)
```

- g) Si tenemos n argumentos, tal como se ve en el caso de plegar que n = 3 y se crean 3 funciones, debemos entonces crear n funciones respectivamente ya que de lo contrario tendría un error de incompletitud.

- h) En el Preludio de Haskell tenemos a la función “foldr” que se comparta como el tipo de dato suministrado, del cual concatena elementos de una lista hasta encontrar el vacío en unos de sus ejemplos. Donde la firma de foldr viene dada por Foldable t => (a -> a -> b) -> b -> t a -> b la cual en comparación es muy cercana.

- Parte 3:

  - a) Se trabaja con “Secuencial s” y no “Secuencial” o “Secuencias s a” ya que solo usando “s” podemos implementar y trabajar de manera genérica con diferentes tipos de datos a en el contexto de “Secuencial s”. Claro que si solo instanciando con “Secuencial” no estaríamos especificando los tipos de datos a los cuales se le aplica Secuencial.

  - b)

```haskell
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
```

  - c)

```haskell
return :: a → Secuencial s a
return value = Secuencial (\x → (value, x))
```

  - d)

```haskell
(Secuencial programa) >>= transformador =
    Secuencial $ \estadoInicial →
        let (resultado, nuevoEstado) = programa estadoInicial
            (Secuencial nuevoPrograma) = transformador resultado
        in nuevoPrograma nuevoEstado
```

- e)

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

return a >>= h

== -- < Aplicando definicion de return >

Secuencial (\x -> (a, x)) >>= h

== -- < Aplicando definicion de (>>=) >

Secuencial $ \y ->
    let (r, ny) = (\x -> (a, x)) y
        (Secuencial nf) = h r
    in nf ny

== -- < Aplicando funcion lanbda >

Secuencial $ \y ->
    let (r, ny) = (a, y)
        (Secuencial nf) = h r
    in nf ny

== -- < Aplicando sustitucion textual de r y ny >

Secuencial $ \y ->
    let (Secuencial nf) = h a
    in nf y

== -- < Como h a toma un tipo a y retorna un
secuencial de tipo diferente, entonces podemos
aplicar que nf = h a textualmente >

Secuencial $ \y -> h a y

== -- < Como h a ya retorna un secuencial entonces
decir que retornamos un secuencial con argumento y
es redundante >

h a
```

```
* Identidad a derecha: m >>= return == m

m >>= return

== -- < Aplicando definicion de (>>=) >

Secuencial $ \y ->
    let (r, ny) = m y
        (Secuencial nf) = return r
    in nf ny

== -- < Aplicando definicion de return >

Secuencial $ \y ->
    let (r, ny) = m y
        (Secuencial nf) = Secuencial (\z -> (r, z))
    in nf ny

== -- < Sustituyendo m por Secuencial >

Secuencial $ \y ->
    let (r, ny) = Secuencial y
        (Secuencial nf) = Secuencial (\z -> (r, z))
    in nf ny

== -- < Como Secuencial es una funcion que retorna un
tupla cuyo primera componente un valor y la segunda
componente es el estado que se pasa como argumento >

Secuencial $ \y ->
    let (r, ny) = (a, y)
        (Secuencial nf) = Secuencial (\z -> (r, z))
    in nf ny

== -- < Aplicando sustitucion textual de r y ny >

Secuencial $ \y ->
    let (Secuencial nf) = Secuencial (\z -> (a, z))
    in nf y

== -- < Aplicando sustitucion textual de nf , cuya
justificacion viene dada la aplicacion de secuencial a
funciones>

Secuencial $ \y ->
    in (\z -> (a, z)) y

== -- < Aplicando funcion lambda >

Secuencial $ \y -> (a, y)

== -- < Como \y -> (a, y) es lo mismo que genera m >

m
```

Para la tercera ley monádica, aplicamos una estrategia de realizar un PRE desarrollo de ambos miembros de la igualdad, a continuación:

```
* Desarrollando un poco la parte izquierda

(m >>= g) >>= h

== -- < Aplicando definicion de >>= sobre m y g>

(Secuencial $ \y →
    let (r, ns) = m y
        (Secuencial np) = g r
    in np ns) >>= h

== -- < Aplicando definicion de >>= sobre Secuencial y h >

Secuencial $ \x →
    let (r', ns') = (\y → let (r, ns) = m y
                              (Secuencial np) = g r
                          in np ns) x
        (Secuencial np') = h r'
    in np' ns'
```

```
* Desarrollando un poco la parte derecha

m >>= (\x → g x >>= h)

== -- < Aplicando definicion de >>= sobre m y lambda func>

Secuencial $ \y →
    let (r, ns) = m y
        (Secuencial np) = (\x → g x >>= h) r
    in np ns

== -- < Aplicando funcion lambda >

Secuencial $ \y →
    let (r, ns) = m y
        (Secuencial np) = g r >>= h
    in np ns

== -- < Aplicando definicion de >>= sobre g r y h >

Secuencial $ \y →
    let (r, ns) = m y
        (Secuencial np) = (Secuencial $ \x →
                              let (r', ns') = g r x
                                  (Secuencial np') = h r'
                              in np' ns')
    in np ns
```

Esto se realiza con la finalidad de observar cierto detalle puntual, sabemos que una prueba por igualdad es demostrar que ambas partes son equivalentes en esencia, en este sentido si notamos las similitudes y diferentes que tienen ambos PRE desarrollos vemos que un lado claro y fundamental es que la variable x e y de la "Secuencial" principal de ambas partes son iguales, al fin y al cabo, es una etiqueta, asi x = y para las secuenciales principales.

Luego, observando en este sentido otras diferencias claves para probar la igualdad es que

```
1) m = (\y → let (r, ns) = m y
                 (Secuencial np) = g r
             in np ns)


2) h r = (Secuencial $ \x →
             let (r', ns') = g r x
                 (Secuencial np') = h r'
             in np' ns')
```

Al probar ambas igualdades entonces probamos que ambos subdesarrollos son iguales lo que equivale a demostrar la tercera ley monádica. Dichas pruebas están a continuación:

```
* Prueba de parte 1

(\y → let (r, ns) = m y
          (Secuencial np) = g r
      in np ns)

== -- < Como m es un secuencia al que le pasamos y entonces
retorna una tupla >

(\y → let (r, ns) = (a, y)
          (Secuencial np) = g r
      in np ns)

== -- < Aplicando sustitucion textual >

(\y → let (Secuencial np) = g a
      in np y)

== -- < Por igualdad de Secuenciales np = g a, aplicando
sustitucion >

(\y → g a y)

== -- < Como g es un funcion que toma un tipo a y devuelve
un secuencial de un tipo b, con retorno y.

(\y → (b, y))

== -- < La funcion  (\y → (b, y)) en esencia es el retorno
que se espera de m >

m
```

```
* Prueba de parte 2

Secuencial $ \x →
        let (r', ns') = g r x
            (Secuencial np') = h r'
        in np' ns'

≡ -- < Como g es un secuencial que recibe tipo r y estado
x como argumentos >

Secuencial $ \x →
        let (r', ns') = (r, x)
            (Secuencial np') = h r'
        in np' ns'

≡ -- < Aplicamos sustitucion textual >

Secuencial $ \x →
        let (Secuencial np') = h r
        in np' x

≡ -- < Por igualdad de Secuenciales np' = h r, aplicando
sustitucion >

Secuencial $ \x → h r x

≡ -- < Como h r ya retorna un secuencial entonces decir
que retornamos un seciencial como argumento y es redundante
>

h r
```

En conclusión, hemos probado parte 1 y parte 2 que verificar la igualdad de los PRE desarrollos de las partes provenientes de la tercera ley monádica, por tanto, queda demostrado.

INVESTIGACION

- Parte 1:

  - a) Se tiene encuenta que las lambdas expresiones pueden tomarse como identificadores entonces eval (E) = E. Así, la prueba queda:

```
(λx . λy . x y y) (λz . z O) L

≡ < Aplicando: eval( (E F) G ) = eval (eval(E F) eval(G) ) >

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

≡ < Aplicando: eval( (λx . E) F ) = eval(E)[x := eval(F)] >

eval( eval( (λy . eval(λz . z O) y y) ) eval(L) )

≡ < Aplicando: Simplificar Evals >

eval( (λy . eval(λz . z O) y y) L )

≡ < Aplicando: eval( (λx . E) F ) = eval(E)[x := eval(F)] >

eval( eval(eval(λz . z O) eval(L) eval(L)) )

≡ < Aplicando: Simplificar Evals >

eval( (λz . z O) L L )

≡ < Aplicando: eval( (E F) G ) = eval (eval(E F) eval(G) ) >

eval( eval( (λz . z O ) L ) eval(L) )

≡ < Aplicando: eval( (λx . E) F ) = eval(E)[x := eval(F)] >

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

≡ < Aplicando: Simplificar Evals >

L O L
```

- b) Por definición se dice que las aplicaciones funcionales son no conmutativas, es decir que el orden de evaluar E y F. siendo ellas expresiones lambdas. es relevante. Si definimos E = (λy. X y) y F = (λx. x D) al realizar E F tenemos como resultado “X (λx. x D)”, en el otro caso si tenemos F E da como resultado “X D” lo que muestra la falla de aplicaciones funcionales en el aspecto conmutativo

- c) Recordando la problemática que se tiene en Lógica Simbólica cuando se tienen variables libres que se generaban al no estar presentan en la definición de cuantificadores y la estrategia de renombrar variables para evitar la confusión y tener un ambiente claro en variables. Pues una estrategia será eso, redefinimos mediante la sustitución por equivalencia, es decir (λy. x y) == (λz. x z) lo que resolvería la problemática.

- Parte 2:

- a)

```
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)

subs (id const) const id

==

(id const) id (const id)

==

const id (const id)

==

id
```

- b) Esta pregunta se responde basado en lo investigado de combinadores de SKI y cierta aplicación de lambda cálculos:
  - En combinadores de SKI se tiene SII(SII) = I(SII)(I(SII)) = SII(I(SII)) = SII(SII)
  - Por lambda cálculos tenemos que si consideramos la expresión (λx . x x) (λx . x x) = (λx . x x) (λx . x x) presenta un bucle.

PD: Puede notar que incluyo conceptos de SKI (que debe ser investigado en la 2.D) dado que esta pregunta es la última que respondo por ser la mas difícil de argumentar. Pues la problemática viene en la firma de subs para poder tener una expresión en bucle.

Entonces, a lo único que pude llegar "ignorando" la firma de subs es *subs id id* (*subs id id*) que sigue la misma regla de SKI y de cierta forma las expresiones de lambda calculo mostrados. En este sentido, tal como se ve en la imagen obtenemos el bucle deseado. En mi mente, la 2.D me daba la solución por lo visto de "SII(SII)" suponiendo que S = subs e I = id, entonces "debería" dar lo mismo.

```
subs id id (subs id id)

==

[id (subs id id)] (id (subs id id))

==

subs id id (subs id id)
```

- c) Para modificar id usando const y subs se propone "subs const () x" donde x es el valor dado, veamos la siguiente prueba:

```
subs const () x

==

const x (() x)

==

x
```

- d) Primero tengamos claro que son los combinadores de SKI. El cálculo de combinadores SKI es un sistema de lógica combinatoria y un sistema computacional que puede considerarse como un lenguaje de programación. Este sistema se compone de tres combinadores fundamentales: S, K e I. Estos combinadores se utilizan para expresar cualquier función en términos de ellos mismos, lo que demuestra la capacidad de expresividad del sistema.

  Combinadores SKI:
    - S: Representa la aplicación de una función a otra función y luego aplicar el resultado a una tercera función.
    - K: Representa una función constante que toma dos argumentos y devuelve el primero.
    - I: Representa la identidad, es decir, toma un argumento y lo devuelve sin modificarlo.

  En este sentido podemos decir entonces que la relación entre id, const y subs es que S = subs, K = const e I = id. Donde K e I son relativamente directas de explicar, sin embargo, para subs es una función que toma tres argumentos: una función que toma dos argumentos y devuelve un tercero, una función que toma un argumento y devuelve otro, y un valor. La función subs aplica la primera función al tercer argumento y al resultado de aplicar la segunda función al tercer argumento. En otras palabras, subs combina las dos funciones de manera que la primera función recibe como argumentos el valor y el resultado de aplicar la segunda función al valor, lo que define un caso particular de S.