

Tarea II: Haskell (10 pts)

Implementación:

0. (3.5 pts) – Considere el tipo de datos `ArbolMB`, que se define a continuación:

```
data ArbolMB a = Vacio
               | RamaM a (ArbolMB a)
               | RamaB a (ArbolMB a) (ArbolMB a)
```

- a) (0.25 pts) – Los constructores de un tipo de datos pueden verse como funciones que reciben (currificados) los argumentos original del mismo y arrojan como resultado un valor del tipo en cuestión.

Por ejemplo:

```
RamaM :: a -> ArbolMB a -> ArbolMB a
```

Diga los tipos correspondientes a los constructores `Vacio` y `RamaB`, vistos como funciones.

- b) (0.25 pts) – Se desea implementar una función que transforme valores de tipo `(ArbolMB a)` en algún otro tipo `b`. Claramente, dicha función debe tener tres casos (uno por cada constructor). Implementaremos cada caso como una función aparte: `transformarVacío`, `transformarRamaM` y `transformarRamaB` respectivamente. Cada una de estas funciones debe tomar los mismos argumentos que los constructores respectivos. Sin embargo, la transformación se hará a profundidad, por lo que se puede suponer que cada argumento de tipo `(ArbolMB a)` ya ha sido transformado a `b`.

Por ejemplo:

```
transformarRamaM :: a -> b -> b
```

Diga los tipos correspondientes a las funciones transformadoras `transformarVacío` y `transformarRamaB`.

- c) (1 pt) – Implementaremos ahora la función transformadora deseada, tomando como argumentos las tres funciones que creamos en la parte (b). Llamaremos a esta función `plegarArbolMB` y su firma (la cual debe completar con su respuesta a la parte (b)) sería la siguiente:

```
plegarArbolMB :: ( ? )      -- El tipo de transformarVacío
-> (a -> b -> b) -- El tipo de transformarRamaM.
-> ( ? )      -- El tipo de transformarRamaB.
-> ArbolMB a   -- El árbol a plegar.
-> b           -- El resultado del plegado.
```

Complete la definición de la función `plegarArbolMB`, propuesta a continuación, recordando que las transformaciones deben hacerse a profundidad para poder garantizar un valor transformado como argumento a las diferentes funciones (*nótese que la función auxiliar `plegar` recibe implícitamente el valor de tipo `(ArbolMB a)` a considerar*).

```
plegarArbolMB transVacio transRamaM transRamaB = plegar
  where
    plegar Vacio          = ¿?
    plegar (RamaM x y)    = transRamaM x (plegar y)
    plegar (RamaB x y z) = ¿?
```

- d) (0.5 pts) – Usando nuestra función `plegarArbolMB`, se desea implementar ahora una función `sumaArbolMB`, que dado un valor de tipo `(Num a) => ArbolMB a` calcule y devuelva la suma de todos datos almacenados en el tipo.

Complete la definición de la función `sumaArbolMB`, propuesta a continuación, usando únicamente una llamada a `plegarArbolMB` y definiendo las funciones de transformación necesarias. (*nótese que la función `plegarArbolMB` recibe implícitamente el valor de tipo `((Num a) => ArbolMB a)` a considerar*).

```
sumaArbolMB :: (Num a) => ArbolMB a -> a

sumaArbolMB = plegarArbolMB transVacio transRamaM transRamaB
  where
    transVacio = ¿?
    transRamaM = ¿?
    transRamaB = ¿?
```

- e) (0.5 pts) – Usando nuevamente nuestra función `plegarArbolMB`, se desea implementar ahora una función `aplanarArbolMB`, que dado un valor de tipo `ArbolMB a` calcule y devuelva una sola lista con todos los elementos contenidos en la estructura. En el caso de la rama con un solo hijo, el elemento debe ir antes que los elementos del hijo. En el caso de la rama con dos hijos, los elementos del primer hijo deben ir antes que el elemento de la rama y este, a su vez, debe ir antes que los elementos del segundo hijo (un recorrido en *in-order*).

Complete la definición de la función `aplanarArbolMB`, propuesta a continuación, usando únicamente una llamada a `plegarArbolMB` y definiendo las funciones de transformación necesarias. (*nótese que la función `plegarArbolMB` recibe implícitamente el valor de tipo `(ArbolMB a)` a considerar*).

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

aplanarArbolMB = plegarArbolMB transVacio transRamaM transRamaB
  where
    transVacio = ¿?
    transRamaM = ¿?
    transRamaB = ¿?
```

- f) (0.5 pt) – Usando nuevamente nuestra función `plegarArbolMB`, se desea implementar ahora una función `analizarArbolMB`, que dado un valor de tipo `(Ord a) => ArbolMB a` calcule y devuelva *posiblemente* una tupla con 3 elementos. El primero debe ser el mínimo elemento presente en la estructura, el segundo debe ser el máximo y el tercero debe ser un booleano que sea cierto si y solo si la lista que resultaría de llamar a la función `aplanarArbolMB` estaría ordenada de menor a mayor (esto es, que sea un árbol de búsqueda). En el caso de un valor de tipo `Vacio`, se debe devolver el valor `Nothing`. (*Pista: No es conveniente llamar explícitamente a la función `aplanarArbolMB` para calcular el 3er elemento de la tupla.*)

Complete la definición de la función `analizarArbolMB`, propuesta a continuación, usando únicamente una llamada a `plegarArbolMB` y definiendo las funciones de transformación necesarias. (*nótese que la función `plegarArbolMB` recibe implícitamente el valor de tipo `((Ord a) => ArbolMB a)` a considerar.*)

```
analizarArbolMB :: (Ord a) => ArbolMB a -> Maybe (a, a, Bool)

analizarArbolMB = plegarArbolMB transVacio transRamaM transRamaB
  where
    transVacio = ¿?
    transRamaM = ¿?
    transRamaB = ¿?
```

- g) (0.25 pts) – Considere ahora un tipo de datos más general `Gen a`, con n constructores diferentes. ¿Si se quisiera crear una función `plegarGen`, con un comportamiento similar al de `plegarArbolMB`, cuantas funciones debe tomar como argumento (además del valor de tipo `Gen a` que se desea plegar)?

- h) (0.25 pts) – Considere ahora el caso especial donde hay 2 posibles constructores.

```
data [a] = (:) a [a]
         | []
```

¿Que función predefinida sobre listas, en el Preludio de Haskell, tiene una firma y un comportamiento equivalente al de implementar una función de plegado para el tipo propuesto?

- i) (Extra) – ¿Qué `typeclass` de Haskell incluye a las estructuras que permiten este comportamiento?

1. (3.5 pts) – Los monads son estructuras que representan cálculos con algún comportamiento particular, encapsulando la implementación del mismo en las definiciones de sus funciones `>>=` y `return`. Por ejemplo: el monad `Maybe` representa cálculos que pueden fallar, el monad `[]` representa cálculos no-deterministas y el monad `IO` representa cálculos impuros.

Se desea implementar entonces un monad que represente cálculos secuenciales. Es decir, dado un estado inicial (por ejemplo, valor de variables en el alcance) se debe obtener un resultado final para el cálculo y un nuevo estado (resultado de posibles alteraciones al estado inicial). Notemos entonces que un cálculo secuencial en realidad puede verse como un alias para una función `s -> (a, s)`, donde `s` es el tipo del estado y `a` el tipo del resultado.

Construyamos un tipo de datos entonces para representar cálculos secuenciales.

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

De la definición anterior debemos notar dos cosas: el identificador `Secuencial` es usado tanto como nombre de tipo como constructor; la notación `newtype` se ha utilizado pues solo existe un posible constructor con un solo argumento. Por lo tanto, dicho argumento es equivalente en contenido al tipo completo, pero conviene no hacerlo un alias para que los tipos no se mezclen (no pasar funciones cualesquiera como cálculos secuenciales). Queremos que nuestro tipo sea un monad, por lo que haremos una instancia para él.

```
instance Monad (Secuencial s) where ...
```

- a) (0.5 pts) – ¿Por qué se tomó `(Secuencial s)` como la instancia para el monad y no simplemente `Secuencial`?
- b) (1 pt) – Diga las firmas para las funciones `return`, `>>=`, `>>` y `fail` para el caso especial del monad `(Secuencial s)`
- c) (1 pt) – Implemente la función `return` de tal forma que *inyecte* el argumento pasado como argumento, dejando el estado inicial intacto. Esto es, dado un estado inicial, el resultado debe ser el argumento pasado junto al estado inicial sin cambios.
- d) (1 pt) – Complete la implementación de la función `>>=` que se da a continuación:

```
(Secuencial programa) >>= transformador =  
  Secuencial $ \estadoInicial ->  
    let (resultado, nuevoEstado) = programa      ¿?  
        (Secuencial nuevoPrograma) = transformador ¿?  
    in nuevoPrograma ¿?
```

(Pista: Ayúdese con los tipos esperados y la intuición para dar un valor a cada una de las interrogantes, las cuales corresponderán cada una a un solo identificador de los previamente definidos.)

- e) (Extra) – ¿Conoce algún *monad* de Haskell que tenga un comportamiento similar? ¿Cuál?

Investigación:

(3 pts) – Considere las siguientes funciones:

```
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)
```

La primeras dos funciones son parte del preludio de Haskell.

- a) (0.5 pts) – Evalúe la expresión: `subs (id const) const id`. No evalúe la expresión en Haskell, pues si el resultado es una función no podrá imprimirlo. Evalúe la expresión a mano y exponga el resultado en término de las funciones antes propuestas (utilice evaluación normal: primero la función luego los argumentos).
- b) (0.5 pts) – Proponga una expresión (únicamente compuesta por las funciones definidas anteriormente), que no esté en forma normal, cuya evaluación resulte en la misma expresión y por lo tanto nunca termine.
- c) (1 pt) – Reimplemente la función `id` en términos de `const` y `sub`. (*Pista: puede utilizar el tipo unitario () para representar un argumento del cual no importa su valor, pero que igual debe ser pasado como parámetro a una función.*)
- d) (1 pt) – Discuta la relación entre las funciones propuestas y el cálculo de combinadores SKI.

Detalles de la Entrega

La entrega de la tarea consistirá de un único archivo `t2_<carné1>&<carné2>.pdf`, donde `<carné1>` y `<carné2>` son los números de carné de los integrantes de su equipo. Tal archivo debe ser un documento PDF con su implementación para las funciones pedidas y respuestas para las preguntas planteadas. Por ejemplo, si el equipo está conformado por 00-00000 y 11-11111, entonces su entrega debe llamarse: `t2_00-00000&11-11111.pdf`.

La tarea deberá ser entregada a ambos profesores encargados del curso (Carlos Infante y Alexander Romero), únicamente a sus direcciones de correo electrónico institucionales: (13-10681@usb.ve y 13-11274@usb.ve) a más tardar el Lunes 24 de Febrero, a las 11:59pm. VET.