CI3661 – Laboratorio de Lenguajes de Programación
Grupo 19 03-36048 María Gracia Hidalgo
05-38978 Alejandro Tarazona

Tarea 2: Haskell

Pregunta 0.

a) Diga los tipos correspondientes a los constructores Papel, Valle y Compuesto, vistos como funciones.

```
Papel :: b
Valle :: a -> Origami a -> Origami a
Compuesto :: Origami a -> Origami a
```

b) Diga los tipos correspondientes a las funciones transformadoras transformarPapel, transformarValle y transformarCompuesto

```
plegarOrigami :: (b)
                 -> (a -> b -> b)
                 -> (a -> b -> b)
                 -> (b -> b -> b)
                 -> Origami a
                 -> b
```

c) Complete la definición de la función plegarOrigami

```
plegarOrigami transPapel transPico transValle transCompuesto = plegar
  where
    plegar Papel = transPapel
    plegar (Pico x y) = transPico x (plegar y)
    plegar (Valle x y) = transValle x (plegar y)
    plegar (Compuesto x y) = transCompuesto (plegar x) (plegar y)
```

d) Complete la definición de la función sumarOrigami

```
sumarOrigami :: (Num a) => Origami a -> a
sumarOrigami = plegarOrigami transPapel transPico transValle transCompuesto
  where
    transPapel = 0
    transPico = (+)
    transValle = (+)
    transCompuesto = (+)
```

e) Complete la definición de la función aplanarOrigami

```
aplanarOrigami :: Origami a -> [a]
aplanarOrigami = plegarOrigami transPapel transPico transValle transCompuesto
  where
    transPapel = []
    transPico = (:)
    transValle = (:)
    transCompuesto = (++)
```

f) Complete la definición de la función analizarOrigami

```
analizarOrigami :: (Ord a) => Origami a -> Maybe(a, a, Bool)
analizarOrigami = plegarOrigami transPapel transPico transValle transCompuesto
  where
    transPapel = Nothing
    transPico = (\elem resto ->
      case resto of
        Nothing -> Just(elem, elem, True)
          Just (minimo, maximo, orden) -> Just(min elem minimo, max elem maximo, orden
&& elem <= minimo))
    transValle = (\elem resto ->
      case resto of
        Nothing -> Just(elem, elem, True)
          Just (minimo, maximo, orden) -> Just(min elem minimo, max elem maximo, orden
&& elem <= minimo))
    transCompuesto = (\resto1 resto2 ->
      case (resto1, resto2) of
        (Nothing, Nothing) -> Nothing
        (Nothing, (Just (minimo, maximo, orden))) -> Just (minimo, maximo, orden)
        (Just (minimo, maximo, orden), Nothing) -> Just (minimo, maximo, orden)
          (Just (minimo1, maximo1, orden1), Just (minimo2, maximo2, orden2)) -> Just (min
minimo1 minimo2, max maximo1 maximo2, orden1 && orden2 && maximo1 <= maximo2))
```

g) 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 plegarOrigami, cuántas funciones debe tomar como argumento (además del valor de tipo Gen que se desea plegar)?

plegarGen debe tomar como argumentos tantas funciones como constructores tenga, pues deben considerarse todos los posibles casos de Gen.

h) Considere ahora el caso especial donde hay 2 posibles constructores.

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

```
| []
```

¿Qué 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?

La función foldr tiene un comportamiento similar al del tipo propuesto.

Pregunta 1.

a) ¿Por qué se tomó (Imperativo s) como la instancia para el monad y no simplemente Imperativo?

Los elementos que son instancia de Monad reciben exactamente un parámetro de tipo para producir un tipo concreto. Se utiliza el nombre s para el atributo que es instancia de Monad, para tener una manera genérica de referirse a él. En esta definición, s recibe tipos como parámetros cuando se escribe algo del tipo s a o s b, de manera que s debe transformar un tipo concreto en otra cosa. El atributo s tiene que tomar un tipo concreto como parámetro y el "resultado" de esa aplicación debe ser un tipo concreto. De manera que s tiene que tener el kind <<* -> *>>, que es el kind de las cosas que pueden ser instancia de Monad.

b) Diga las firmas para las funciones return, >>=, >> y fail para el caso especial del monad (Imperativo s)

```
return :: a -> Imperativo s a

>>= :: Imperativo s a -> (a -> Imperativo s b) -> Imperativo s b

>> :: Imperativo s a -> Imperativo s b -> Imperativo s b

fail :: String -> Imperativo s a
```

c) 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.

```
return valor = Imperativo (\ inicial -> (valor, inicial))
```

d) Complete la implementación de la función >>= que se da a continuación:

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

e) Demuestre que las tres leyes monádicas se cumplen para el monad (Imperativo s).

Ley 1: return x >>= f = f x

```
    return a `transformador` f
==>
    \estadoInicial -> let (a, estadoInicial') = (return a) estadoInicial
              m'      = f a
          in  m' estadoInicial'
==>
    \estadoInicial -> let (a, estadoInicial') = (\estadoInicial -> (a, estadoInicial)) estadoInicial
          in  (f a) estadoInicial'
==>
    \estadoInicial -> let (a, estadoInicial') = (a, estadoInicial)
          in  (f a) estadoInicial'
==>
    \estadoInicial -> (f a) estadoInicial
==>
    f a
```

Ley 2: m >>= return = m

```
    f `transformador` return
==>
    \estadoInicial -> let (a, estadoInicial') = f estadoInicial
          in  (return a) estadoInicial'
==>
    \estadoInicial -> let (a, estadoInicial') = f estadoInicial
          in  (\estadoInicial -> (a, estadoInicial)) estadoInicial'
==>
    \estadoInicial -> let (a, estadoInicial') = f estadoInicial
          in  (a, estadoInicial')
==>
    \estadoInicial -> f estadoInicial
==>
    f
```

Ley 3: (f >>= g) >> = h ≡ f >>= (\x -> g x >>= h)

```
    \estadoInicial -> let (a, estadoInicial') = f estadoInicial
          in  (\x -> g x `transformador` h) a estadoInicial'
==>
    \estadoInicial -> let (a, estadoInicial') = f estadoInicial
          in  (g a `transformador` h) estadoInicial'
==>
    \estadoInicial -> let (a, estadoInicial') = f estadoInicial
```

```
        in  (\estadoInicial' -> let (b, estadoInicial'') = g a
                in  h b estadoInicial'') estadoInicial'
==>
    \estadoInicial -> let (a, estadoInicial') = f estadoInicial
            (b, estadoInicial'') = g a estadoInicial'
            (c, estadoInicial''') = h b estadoInicial''
        in  (c, estadoInicial''')
```

Investigación

a) Evalúe la expresión: subs (id const) subs const.

subs (id const) subs cons
≡
subs const subs const
≡
const subs subs const
≡
const

b) Proponga una expresión cuya evaluación resulte en la misma expresión y por lo tanto, nunca termine.

Si consideramos que

subs id id (x)
≡
id x id x
≡
x x

Entonces, la estructura propuesta sería subs id id (subs id id)

c) Reimplemente la función id en términos de const y sub.

subs const const x
≡
const x const x
≡
const x x
≡
x

d) Discuta la relación entre las funciones propuestas y el cálculo de combinadores SKI.

Las funciones propuestas son equivalentes a los combinadores SKI, con la siguiente

correspondencia:

- S ≡ subs
- K ≡ const
- I ≡ id

Debido a esta equivalencia, todas las propiedades que se cumplen para los combinadores SKI, se cumplen también para las funciones definidas en este segmento.