# Tarea 2

## *Tipos algebraicos y razonamiento ecuacional*

**Ejercicio 1** **25 Pt**

Una *fracción continua* finita a coeficientes enteros es una expresión de alguna de las siguientes formas:

$$a_0, \quad a_0 + \frac{b_o}{a_1}, \quad a_0 + \frac{b_0}{a_1 + \frac{b_1}{a_2}}, \quad a_0 + \frac{b_0}{a_1 + \frac{b_1}{a_2 + \frac{b_2}{a_3}}}, \quad a_0 + \frac{b_0}{a_1 + \frac{b_1}{a_2 + \frac{b_2}{a_3 + \frac{b_3}{a_4}}}}, \quad \ldots$$

donde $a_i, b_i \in \mathbb{Z}$. La primera expresión es una fracción continua de grado 0, la segunda de grado 1, y así sucesivamente. Es fácil ver que una fracción continua es o bien una fracción continua "simple", de la forma $a_0 \in \mathbb{Z}$, o bien una fracción continua "compuesta", de la forma $a_i + \frac{b_i}{d}$ donde $a_i, b_i \in \mathbb{Z}$ y $d$ es otra fracción continua.

(a) [3 Pt] Declare el tipo de datos recursivo `ContFraction` con los constructores `Simple` y `Compound` (de acuerdo a lo arriba descrito). A manera de ejemplo, bajo su declaración la fracción continua $t = 3 + \frac{1}{4 + \frac{1}{12 + \frac{1}{4}}}$ se representaría como:

```
t :: ContFraction
t = (Compound 3 1 (Compound 4 1 (Compound 12 1 (Simple 4))))
```

Para la declaración de `ContFraction` utilice enteros de precisión arbitraria (`Integer`).

(b) [2 Pt] Defina la función `evalCF :: ContFraction -> (Integer, Integer)` que evalúa una fracción continua, devolviendo el número racional que representa, expresado en forma de par de enteros, donde el primer componente del par representa el numerador, y el segundo el denominador.

```
> evalCF t
(649,200)
```

No hace falta simplificar la fracción que se devuelve.

(c) [2 Pt] Defina la función `degree :: ContFraction -> Integer` que devuelve el grado de una fracción continua (arriba definido).

(d) [5 Pt] Defina (y dé el tipo de) la función `fold` que captura el esquema de recursión primitiva sobre `ContFraction`.

(e) [6 Pt] Redefina las funciones `evalCF` y `degree` usando el `fold` arriba definido (en vez de una recursión explícita).

(f) [4 Pt] Por último, defina la función `frac2ConFrac :: (Integer, Integer) -> ContFraction` que transforma un número racional no-negativo en su representación en forma de fracción continua.

```
> frac2ConFrac (649,200)
Compound 3 1 (Compound 4 1 (Compound 12 1 (Simple 4)))
```

Para ello, siga el algoritmo descrito en este enlace.

(g) [3 Pt] Usando QuickCheck verifique que $\forall m, n \in \mathbb{Z}$, si $m, n > 0$, entonces la fracción devuelta por $eval(frac2ConFrac(m, n))$ coincide con la fracción representada por $(m, n)$.

**Ejercicio 2** **13 Pt**

En este ejercicio adaptaremos el chequeador de tautologías visto en la clase 12.

(a) [4 Pt] Dar el tipo y definir la función `foldF` que captura el esquema de recursión primitiva asociado al tipo recursivo `Formula`. Con respecto al orden de los argumentos de `foldF`, seguir el mismo orden en el que aparecen los constructores de `Formula` en su declaración.

(b) [4 Pt] Redefinir las funciones `eval`, `fvar` utilizando `foldF`. Cuando defina `eval` puede asumir que siempre que evaluemos una fórmula lo vamos a hacer sobre una valuación que le asigna un único valor a cada variable (i.e. no es necesario implementar ningún tratamiento especial de errores). Para la definición de `eval` (en el caso de variables) debe usar la función `find` del Ejercicio 3.

(c) [5 Pt] Definir una función `isTaut :: Formula -> Maybe Valuation` que devuelva `Nothing` si la fórmula es una tautología, y `Just v` si la formula no es una tautología, donde `v` es una valuación (cualquiera si hay más de una) que haga la fórmula falsa.

```
> isTaut (Imply (And (Var 'A') (Imply (Var 'A') (Var 'B'))) (
    Var 'B'))
> Nothing

> isTaut (Imply (Var 'A') (And (Var 'A') (Var 'B')))
> Just [('A',True),('B',False)]
```

**Ejercicio 3** **5 Pt**

En clase vimos que podemos usar el constructor de tipos `Maybe` para definir funciones parciales. La limitación que tiene este constructor de tipos es que en caso de que haya múltiples causas de error, no permite distinguirlas (todas son mapeadas a `Nothing`). Para estos casos podemos usar en vez el constructor de tipos

```
data Either a b = Left a | Right b
```

provisto por el preludio estándar: usamos el constructor de valores `Left` para denotar un error y el constructor de valores `Right` para denotar un resultado exitoso.

(a) [4 Pt] Dados los siguientes sinónimos de tipos para representar errores y tablas asociativas

```
type Assoc k v = [(k,v)]
type Error = String
```

se pide definir la función de look-up

```
find :: (Eq k,Show k,Eq v) => k -> Assoc k v -> Either Error
   v
```

que permite distinguir el origen de los errores: clave no encontrada o clave asociada a múltiples valores:

```
> find 2 [(2,'a'),(1,'b')]
> Right 'a'

> find 2 [(2,'a'),(1,'b'), (2,'a')]
> Right 'a'

> find 3 [(2,'a'),(1,'b')]
> Left "Key 3 not found"

> find 2 [(2,'a'),(1,'b'),(2,'c')]
> Left "Multiple values for key 2"
```

Para la definición le puede resultar de utilidad la keyword `case ... of ...` que permite hacer pattern matching en lado derecho de una definición (i.e. en el cuerpo de la función que se está definiendo).

(b) [1 Pt] Explicar por qué se requiere cada uno de los constraints de tipos `Eq k`, `Show k`, `Eq v` en la definición de `find`.

**Ejercicio 4** **6 Pt**

Usando `foldNat` para capturar la recursión, dar una definición de la función `f :: Nat -> Nat` tal que,

$$
\begin{aligned}
f(0) &= 2 \\
f(n) &= \begin{cases} n + f(n-2) & \text{si } n \text{ es par} \\ n^2 + f(n-1) & \text{si } n \text{ es impar} \end{cases} \qquad n \geq 1
\end{aligned}
$$

**Ejercicio 5** **11 Pt**

Considere la siguiente declaración de árboles binarios y del *fold* asociado.

```
data BinTree a = Leaf a | InNode (BinTree a) a (BinTree a)
```

```
foldBT :: (b -> a -> b -> b) -> (a -> b) -> (BinTree a -> b)
foldBT f g (Leaf v)          = g v
foldBT f g (InNode t1 v t2) = f (foldBT f g t1) v (foldBT f g t2)
```

(a) [3 Pt] Sean `f :: b -> a -> b -> b`, `g :: a -> b`, `f' :: b' -> a -> b' -> b'`, `g :: a -> b'` y `h :: b -> b'` tales que

1. `h (g v) = g' v`
2. `h (f x1 v x2) = f' (h x1) v (h x2)`

Usando inducción estructural, probar que `h . foldBT f g = foldBT f' g'`.

(b) [4 Pt] Usar el resultado del item (a) para probar que `length . falttenBT = sizeBT`, donde `flattenBT` y `sizeBT` devuelven la versión aplanada (en forma de lista) y el número de nodos de un árbol:

```
mirrorBT :: BinTree a -> BinTree a
mirrorBT = foldBT (\r1 v r2 -> InNode r2 v r1) Leaf

flattenBT :: BinTree a -> [a]
flattenBT = foldBT (\r1 v r2 -> r1 ++ [v] ++ r2) (:[])

sizeBT :: BinTree a -> Int
sizeBT = foldBT (\r1 v r2 -> r1 + 1 + r2) (const 1)
```

*Hint.* Para su prueba puede asumir que `length (xs ++ ys) = length xs + length ys`.

(c) [4 Pt] Usar el resultado del item (a) para probar que `mirrorBT . mirrorBT = idBT` , donde `mirrorBT` devuelve el "espejo" de un árbol e `idBT` representa la función identidad sobre árboles:

```
mirrorBT :: BinTree a -> BinTree a
mirrorBT = foldBT (\r1 v r2 -> InNode r2 v r1) Leaf

idBT :: BinTree a -> BinTree a
idBT = foldBT InNode Leaf
```