PROGRAMACIÓN FUNCIONAL

Tipos de Datos: Tipos Recursivos

Tipos de Datos

- Tipos algebraicos recursivos
- Listas
- Árboles
- Funciones sobre tipos recursivos
- Lenguaje Imperativo Simple (LIS)
- Evaluación de un programa LIS

Tipos de Datos Recursivos

- Un tipo algebraico recursivo
  - tiene al menos uno de los constructores con el tipo que se define como argumento
  - es la concreción en Haskell de un conjunto definido inductivamente
- Ejemplos:

```
data N = Z | S N
data BE = TT | FF | AA BE BE | NN BE
```

- ¿Qué elementos tienen estos tipos?

Tipos de Datos Recursivos

- Cada constructor define un caso de una definición inductiva de un conjunto.
  - Si tiene al tipo definido como argumento, es un *caso inductivo*, si no, es un *caso base*.
- El pattern matching
  - provee análisis de casos
  - permite acceder a los elementos inductivos que forman a un elemento dado
- Por ello, se pueden definir funciones recursivas

Tipos de Datos Recursivos

- Ejemplo: data N = Z | S N

```
size :: N -> Int
size Z = 0
size (S x) = 1 + size x
addN :: N -> N -> N
addN Z m = m
addN (S n) m = S (addN n m)
```

- ¿Puede probar la siguiente propiedad?

Sean n,m::N finitos, cualesquiera; entonces

```
size (addN n m) = size n + size m
```

Tipos de Datos Recursivos

- Se pueden probar propiedades por inducción estructural (pues representan conjuntos inductivos)
- **Demostración**: por inducción en la estructura de n
  - Caso base: n = Z
    - Usar addN.1, 0 neutro de (+) y size.1
  - Caso inductivo: n = S n'
  - HI: size (addN n' m) = size n' + size m
    - Usar addN.2, size.2, HI, asociatividad de (+), y size.2

Listas

- Una definición equivalente a la de listas

    data List a = Nil | Cons a (List a)

- La sintaxis de listas es equivalente a la de esta definición:
    - [ ] es equivalente a Nil
    - (x:xs) es equivalente a (Cons x xs)
- Sin embargo, (List a) y [a] son tipos distintos

Listas

- Considerar las definiciones

```
(++) :: [a] -> [a] -> [a]
[ ] ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
sum :: [Int] -> Int
sum [ ] = 0
sum (n:ns) = n + sum ns
```

- Demostrar que para todo par xs, ys de listas finitas, vale que:

```
sum (xs ++ ys) = sum xs + sum ys
```

Listas

- **Propiedad**: para todo par de xs, ys de listas finitas
  sum (xs ++ ys) = sum xs + sum ys
- **Demostración**: por inducción en xs
  - Caso base: xs = []
    - Usar (++).1, 0 neutro de (+) y sum.1
  - Caso inductivo: xs = (x:xs')
  - HI: sum (xs' ++ ys) = sum xs' + sum ys
    - Usar (++).2, sum.2, HI, asoc. de (+) y sum.2

Árboles

- Un árbol es un tipo algebraico tal que al menos un elemento compuesto tiene dos componentes inductivas
- Se pueden usar TODAS las técnicas vistas para tipos algebraicos y recursivos
- Ejemplo:

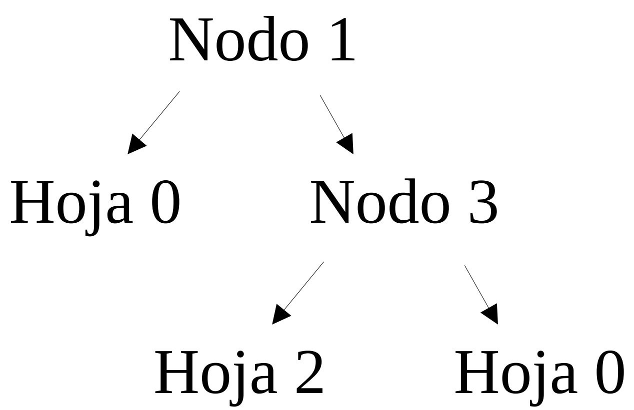
```
data Arbol a = Hoja a | Nodo a (Arbol a) (Arbol a)
```

- ¿Qué elementos tiene el tipo (Arbol Int)?

Árboles

- Si representamos elementos de tipo Arbol Int mediante diagramas jerárquicos

```
aej = Nodo 1 (Hoja 0)
             (Nodo 3 (Hoja 2) (Hoja 0))
```



Árboles

- ¿Cuántas hojas tiene un (Arbol a)?

```
hojas :: Arbol a -> Int
hojas (Hoja x) = 1
hojas (Nodo x t1 t2) = hojas t1 + hojas t2
```

- ¿Y cuál es la altura de un (Arbol a)?

```
altura :: Arbol a -> Int
altura (Hoja x) = 0
altura (Nodo x t1 t2) = 1+ (altura t1 `max` altura t2)
```

- ¿Puede mostrar que para todo árbol finito a, hojas a ≤ 2^(altura a)? ¿Cómo?

Árboles

- ¿Cómo reemplazamos una hoja?
- Ej: Cambiar los 2 en las hojas por 3.

```
cambiar2 :: Arbol Int -> Arbol Int
cambiar2 (Hoja n) = if n==2
                      then Hoja 3
                      else Hoja n
cambiar2 (Nodo n t1 t2) =
         Nodo n (cambiar2 t1) (cambiar2 t2)
```

- ¿Cómo trabaja cambiar2? Reducir (cambiar2 aej)

Árboles

- Más funciones sobre árboles

```
duplA :: Arbol Int -> Arbol Int
duplA (Hoja n) = Hoja (n*2)
duplA (Nodo n t1 t2) =
        Nodo (n*2) (duplA t1) (duplA t2)
sumA :: Arbol Int -> Int
sumA (Hoja n) = n
sumA (Nodo n t1 t2) = n + sumA t1 + sumA t2
```

- ¿Cómo evalúa la expresión (duplA aej)?
- ¿Y (sumA aej)?

Árboles

- Recorridos de árboles

```
inOrder, preOrder :: Arbol a -> [ a ]
inOrder (Hoja n) = [ n ]
inOrder (Nodo n t1 t2) =
        inOrder t1 ++ [ n ] ++ inOrder t2

preOrder (Hoja n) = [ n ]
preOrder (Nodo n t1 t2) =
        n : (preOrder t1 ++ preOrder t2)
```

- ¿Cómo sería posOrder?

Expresiones Aritméticas

- Definimos expresiones aritméticas
  - constantes numéricas
  - sumas y productos de otras expresiones

  ```
  data ExpA = Cte Int | Suma ExpA ExpA
                      | Mult ExpA ExpA
  ```
- Ejemplos:
  - 2 se representa (Cte 2)
  - (4*4) se representa (Mult (Cte 4) (Cte 4))
  - ((2*3)+4) se representa
    Suma (Mult (Cte 2) (Cte 3)) (Cte 4)

Expresiones Aritméticas

- ¿Cómo dar el significado de una ExpA?

```
evalEA :: ExpA -> Int
evalEA (Cte n) = n
evalEA (Suma e1 e2) = evalEA e1 + evalEA e2
evalEA (Mult e1 e2) = evalEA e1 * evalEA e2
```

- Reduzca:

```
evalEA (Suma (Mult (Cte 2) (Cte 3)) (Cte 4))

evalEA (Mult (Cte 2) (Suma (Cte 3) (Cte 4)))
```

Definición de LIS

- Definimos un Lenguaje Imperativo Simple
  - asignación de expresiones numéricas
  - sentencias if y while sobre expresiones booleanas
  - secuencia de sentencias
- Ejemplo:

```
a := n; fac_n := 1;
while (a > 0) do
    fac_n := a * fac_n;
    a := a - 1;
od;
```

Definición de LIS

- Definimos tipos algebraicos para representar un programa LIS

```
type Variable = String

type Program = [Statement]

data Statement = Skip
               | Assign Variable NExp
               | If BExp [Statement] [Statement]
               | While BExp [Statement]
```

Definición de LIS (cont.)

```
data NExp = Vble String | NCte Int | Add NExp NExp
          | Sub NExp NExp | Mul NExp NExp
          | Div NExp NExp   | Mod NExp NExp

data BExp = BCte Bool        | Not BExp
          | And BExp BExp   | Or BExp BExp
          | RelOp ROp NExp NExp

data ROp = Equal            | NotEqual
         | Greater          | Lower
         | GreaterEqual     | LowerEqual
```

Evaluador de LIS

- Tipo abstracto para representar la memoria

```
data State              -- Tipo abstracto de estados
initialState :: State
  -- Un estado sin variables
getVar :: Variable -> State -> Maybe Int
  -- Retorna el valor asociado a la variable dada
putVar :: Int -> Variable -> State -> State
  -- Asigna un valor a una variable
variables :: State -> [ Variable ]
  -- Retorna las variables definidas en el estado
```

Evaluador de LIS (cont.)

- Semántica de expresiones aritméticas

```
evalN :: NExp -> (State -> Int)
evalN (Vble x) st =
        case (getVar x st) of
           Nothing -> error ("variable "++x++" indefinida")
           Just v  -> v
evalN (NCte n) st = n
evalN (Add e1 e2) st = evalN e1 st + evalN e2 st
...
```

- Observar que las expresiones tienen variables, y eso complica su semántica

Evaluador de LIS (cont.)

- Semántica de expresiones booleanas

```
evalB :: BExp -> (State -> Bool)
evalB (BCte b) st = b
evalB (RelOp rop e1 e2) st =
        evalROp rop (evalN e1 st) (evalN e2 st)
evalB (And e1 e2) st = evalB e1 st && evalB e2 st
...
evalROp :: ROp -> Int -> Int -> Bool
evalROp Equal = (==)
...
```

- ¿Por qué hace falta el estado para dar significado a una expresión booleana?

Evaluador de LIS (cont.)

- Semántica de sentencias LIS

```
evalS :: Statement -> (State -> State)

evalS Skip  = \st -> st

evalS (Assign x ne)  = \st -> putVar (evalN ne st) x st

evalS (If be p1 p2) =
        \st -> if (evalB be st) then (evalP p1 st)
                                else (evalP p2 st)

evalS (While be p) =
        \st -> evalS (If be (p ++ [While be p]) [Skip]) st
```

Evaluador de LIS (cont.)

- Semántica de programas LIS

```
evalP :: Program -> (State -> State)
evalP [ ] = \st -> st
evalP (p:ps) =
        \st -> let st' = evalS p st
                in evalP ps st'
```

- ¡Observar cómo la secuencia de sentencias ALTERA el estado antes de proseguir!

Resumen

- Tipos algebraicos recursivos
  - Aleph, BE, Listas,
- Árboles
  - Arbol a, ExpA
- Funciones sobre tipos recursivos
- Programas LIS
- Semántica (integral) de programas LIS