PROGRAMACIÓN FUNCIONAL

Tipos de Datos: Tipos Recursivos

Tipos de Datos

- Tipos algebraicos recursivos
- Funciones sobre tipos recursivos
- Números; Listas; Árboles
- Expresiones aritméticas (sin y con variables)
- Lenguaje Imperativo Simple (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
  - Asignación de significado

```
evalN :: N -> Int
evalN Z     = ...
evalN (S x) = ... evalN x ...
```

  - ¡Usamos recursión estructural!

Tipos de Datos Recursivos

- Ejemplo: data N = Z | S N (cont.)
  - Asignación de significado

```
evalN :: N -> Int
evalN Z      = 0
evalN (S x) = 1 + evalN x
```

  - El tipo N es notación unaria para expresar números enteros (Int)

Tipos de Datos Recursivos

- Ejemplo: data N = Z | S N (cont.)
  - Manipulación simbólica

```
addN :: N -> N -> N
addN Z     m = ...
addN (S n) m = ... (addN n m) ...
```

  - Otra vez usamos recursión estructural

Tipos de Datos Recursivos

- Ejemplo: data N = Z | S N (cont.)
  - Manipulación simbólica

```
addN :: N -> N -> N
addN Z     m = m
addN (S n) m = S (addN n m)
```

  - No hay significado en sí mismo en esta manipulación

Tipos de Datos Recursivos

- Ejemplo: data N = Z | S N (cont.)
  - Coherencia de significación con manipulación
  - ¿Puede probarse la siguiente propiedad?
    Sean n,m::N finitos, cualesquiera; entonces

    evalN (addN n m) = evalN n + evalN m

  - ¿Cómo? ...
  - La propiedad captura el vínculo entre el significado y la manipulación simbólica

Tipos de Datos Recursivos

- Por inducción estructural
  (pues el tipo representa a un conjunto inductivo)
- **Demostración**: por inducción en la estructura de n
  - Caso base: n = Z
    - Usar addN.1, 0 neutro de (+) y evalN.1
  - Caso inductivo: n = S n'
  - HI: evalN (addN n' m) = evalN n' + evalN m
    - Usar addN.2, evalN.2, HI, asociatividad de (+), y evalN.2

Tipos de Datos Recursivos

- Ejemplo: data N = Z | S N (cont.)
  - Más manipulación simbólica

```
prodN :: N -> N -> N
prodN Z     m = Z
prodN (S n) m = addN (prodN n m) m
```

- ¿Puede probar la siguiente propiedad?

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

  evalN (prodN n m) = evalN n * evalN m

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

```
sum :: [Int] -> Int          -- Significado
sum [ ]      = 0
sum (n:ns)   = n + sum ns

(++) :: [a] -> [a] -> [a]    -- Manipulación simbólica
[ ] ++ ys      = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

- Coherencia entre significado y manipulación:
  Demostrar que para todas xs, ys listas finitas vale que:

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

Listas

- **Propiedad**: para todo par 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

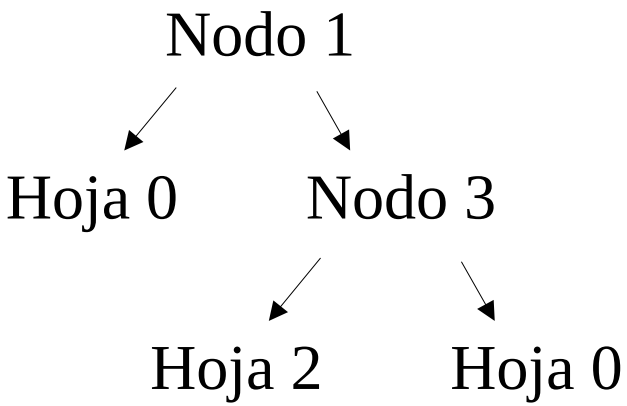
Árboles

- 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)          = ...
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)          = ...
altura (Nodo x t1 t2) = ... altura t1 ... altura t2 ...
```

- Observar el uso de recursión estructural

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 t, hojas t ≤ 2^(altura t)? ¿Cómo?

Árboles

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

```
cambiar2 :: Arbol Int -> Arbol Int
cambiar2 (Hoja n) = …

cambiar2 (Nodo n t1 t2) =
                … (cambiar2 t1) … (cambiar2 t2) …
```

- ¡Más recursión estructural!
- El término “cambiar” es engañoso...

Árboles

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

```
cambiar2 :: Arbol Int -> Arbol Int
cambiar2 (Hoja n) = …

cambiar2 (Nodo n t1 t2) =
                … (cambiar2 t1) … (cambiar2 t2) …
```

- ¡Más recursión estructural!
- El término “cambiar” es engañoso...

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

- ¡Recursión estructural!

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
  - Transformación de un árbol en una lista
    (estructura no lineal a estructura lineal)

```
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?

Árboles

- Recorridos de árboles
  - Transformación de un árbol en una lista
    (estructura no lineal a estructura lineal)

```
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 | Sum 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
    Sum (Mult (Cte 2) (Cte 3)) (Cte 4)

Expresiones Aritméticas

- Definimos expresiones aritméticas
  - alternativa más simbólica...

    data ExpS = CteS N | SumS ExpS ExpS
    | MultS ExpS ExpS
- Ejemplos:
  - 2 se representa (CteS (S (S Z)))
  - comparar con (Cte 2), donde Int representa números como semántica y no como símbolos (como lo hace N)

Expresiones Aritméticas

- ¿Cómo dar el significado de una ExpA?

```
evalEA :: ExpA -> Int
evalEA (Cte n)          = ...
evalEA (Sum e1 e2)      = evalEA e1 ... evalEA e2
evalEA (Mult e1 e2)     = evalEA e1 ... evalEA e2
```

  - Observar el uso de recursión estructural

Expresiones Aritméticas

- ¿Cómo dar el significado de una ExpA?

```
evalEA :: ExpA -> Int
evalEA (Cte n)          = n
evalEA (Sum 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)))
```

Expresiones Aritméticas

- Comparar con la versión más simbólica

```
evalES :: ExpS -> Int
evalES (CteS n)          = evalN n
evalES (SumS e1 e2)      = evalES e1 + evalES e2
evalES (MultS e1 e2)     = evalES e1 * evalES e2
```

  - Se observa el uso de la función de asignación semántica (evalN) a los números representados como N (símbolos)

Expresiones Aritméticas

- ¿Cómo simplificar una ExpA?
  (Solo quitar las sumas de ceros)
  - Manipulación simbólica

```
simplEA :: ExpA -> ExpA
simplEA (Cte n)          = ...
simplEA (Sum e1 e2)      = ... (simplEA e1) ...
                               (simplEA e2) ...
simplEA (Mult e1 e2)     = ... (simplEA e1) (simplEA e2)
```

  - Observar otra vez el uso de recursión estructural

Expresiones Aritméticas

- ¿Cómo simplificar una ExpA?
  Manipulación simbólica

```
simplEA :: ExpA -> ExpA
simplEA (Cte n)          = Cte n
simplEA (Sum e1 e2)      = armarSuma (simplEA e1)
                                     (simplEA e2)
simplEA (Mult e1 e2)     = Mult (simplEA e1) (simplEA e2)

armarSuma (Cte 0) e  = e
armarSuma e (Cte 0)  = e
armarSuma e1 e2      = Sum e1 e2
```

Expresiones Aritméticas

- ¿La manipulación es correcta?
  - Coherencia entre
    significado y manipulación simbólica
  - Expresado mediante la siguiente propiedad

    para todo e se cumple que

    evalEA (simplEA e) = evalEA e

Expresiones con variables

- ¿Cómo agregar variables a las expresiones?
  - Nuevo constructor

```
type Variable = String

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

  - Además agregamos nuevas operaciones

Expresiones con variables

- ¿Y para asignarles significado?
  - Necesitamos conocer el valor de las variables

    evalNExp :: NExp -> (Memoria -> Int)
  - ¡El significado es una función!
  - Observar el uso del alto orden y la currificación
  - ¿Qué es la memoria?

Expresiones con variables

- Tipo abstracto para representar la memoria

```
data Memoria                -- Tipo abstracto de datos
enBlanco :: Memoria
  -- Una memoria vacía, que no recuerda nada
cuantoVale :: Memoria -> Variable -> Maybe Int
  -- Retorna el valor recordado para la variable dada
recordar :: Memoria -> Variable -> Int -> Memoria
  -- Recuerda un valor paraa una variable
variables :: Memoria -> [ Variable ]
  -- Retorna las variables que recuerda
```

Expresiones con variables

- Semántica de expresiones con variables

```
evalN :: NExp -> (Memoria -> Int)
evalN (Vble x)       mem  =
      ... mem … x ...


evalN (NCte n)       mem   = ... n ...
evalN (Add e1 e2)    mem   = evalN e1 mem … evalN e2 mem
        ...
```

- Observar
  - que las variables complican la semántica
  - el uso de la currificación para pasar la memoria

Expresiones con variables

- Semántica de expresiones con variables

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

- Observar
  - Lo mejor es fallar si la variable está indefinida
  - Algunos lenguajes dan otras cosas (0, o “basura”)

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; facn := 1
while (a > 0)
  { facn := a * facn; a := a – 1 }
```

- ¡Solo sintaxis!

Definición de LIS (cont.)

- Definimos tipos algebraicos para representar un programa LIS

```
data Program = P Bloque
type Bloque = [Comando]
data Comando  = Skip
              | Assign Variable NExp
              | If BExp Bloque Bloque
              | While BExp Bloque
```

Definición de LIS (cont.)

- Definimos tipos algebraicos para representar un programa LIS
  - Los constructores carecen de significado
  - Una definición “equivalente” sería

```
data A = B C
type C = [D]
data D  = E
        | F Variable NExp
        | G BExp C C
        | H BExp C
```

Definición de LIS (cont.)

- Usamos las NExp anteriores, y agregamos expresiones booleanas

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

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

Definición de LIS (cont.)

- Usamos las NExp anteriores, y agregamos expresiones booleanas
  - Continuando con la definición “equivalente”

```
data BExp = I Bool     | J BExp
          | K BExp BExp   | L BExp BExp
          | M ROp NExp NExp

data ROp = N   | O
         | P   | Q
         | R   | S
```

Definición de LIS

- ¿Cómo queda el programa de ejemplo?

```
a := n; facn := 1
while (a > 0)
  {  facn := a * facn; a := a – 1 }
```

se expresa como

```
P [ Assign “a” (Vble “n”)
  , Assign “facn” (NCte 1)
  , While (RelOp Greater (Vble “a”) (NCte 0))
      [  Assign “facn” (Mul (Vble “a”) (Vble “facn”))
      ,  Assign “a” (Sub (Vble “a”) (NCte 1))
      ]
  ]
```

Definición de LIS

- ¿Cómo queda el programa de ejemplo?

```
a := n; facn := 1
while (a > 0)
  {  facn := a * facn; a := a – 1 }
```

se expresa “equivalentemente” como

```
B [ F “a” (Vble “n”)
  , F “facn” (NCte 1)
  , H (M P (Vble “a”) (NCte 0))
      [  F “facn” (Mul (Vble “a”) (Vble “facn”))
      ,  F “a” (Sub (Vble “a”) (NCte 1))
      ]
  ]
```

Evaluador de LIS (cont.)

- Semántica de expresiones booleanas

```
evalB :: BExp -> (Memoria -> Bool)
evalB (BCte b) mem = ...
evalB (RelOp rop e1 e2) mem
   = ... rop ... (evalN e1 mem) ... (evalN e2 mem) ...
evalB (And e1 e2) mem
   = ... evalB e1 mem ... evalB e2 mem ...
...
```

- ¿Por qué hace falta la memoria para dar significado a una expresión booleana?

Evaluador de LIS (cont.)

- Semántica de expresiones booleanas

```
evalB :: BExp -> (Memoria -> Bool)
evalB (BCte b) mem = b
evalB (RelOp rop e1 e2) mem
   = evalROp rop (evalN e1 mem) (evalN e2 mem)
evalB (And e1 e2) mem
   = evalB e1 mem && evalB e2 mem
...
```

- Nuevamente, observar el uso de currificación
- ¿Y la función auxiliar evalROp?

Evaluador de LIS (cont.)

- Semántica de expresiones booleanas (cont.)

```
evalROp :: ROp -> (Int -> Int -> Bool)
evalROp Equal     = (==)
evalROp NotEqual  = (!=)
evalROp Greater   = (>)
...
```

- ¡Observar que el significado se define directamente como una función!

Evaluador de LIS (cont.)

- Semántica de programas LIS

```
evalP :: Program -> (Memoria -> Memoria)
evalP (P bloque) = evalBloque bloque

evalBloque [ ] = \mem -> mem
evalBloque (c:cs) =
        \mem -> let mem' = evalCom c mem
                in evalBloque cs mem'
```

- ¡¡El significado es una función!!
- ¡Observar cómo la secuencia de comandos ALTERA la memoria antes de proseguir!

Evaluador de LIS (cont.)

- Semántica de sentencias LIS

```
evalCom :: Comando -> (Memoria -> Memoria)
evalCom Skip  = ...
evalCom (Assign x ne)
        = ... (evalN ne mem) ...
evalCom (If be bl1 bl2)
        = ... (evalB be mem)
                    ... (evalBloque bl1 mem)
                    ... (evalBloque bl2 mem)
evalCom (While be p)
        = ...??
```

Evaluador de LIS (cont.)

- Semántica de sentencias LIS

```
evalCom :: Comando -> (Memoria -> Memoria)
evalCom Skip  = \mem -> mem
evalCom (Assign x ne)
        = \mem -> recordar mem x (evalN ne mem)
evalCom (If be bl1 bl2)
        = \mem -> if (evalB be mem)
                     then (evalBloque bl1 mem)
                     else (evalBloque bl2 mem)
evalCom (While be p)
      = evalCom (If be (p ++ [While be p]) [Skip])
```

¡No es estructural!

Manipulacion simbólica

- ¿Qué pasa con programas como éste?

```
p :: Program
p = P [ Assign “x” (Add (NCte 10) (NCte 7))
      , Assign “y” (Add (Sub (NCte 59) (Var “x”))
                        (Sub (NCte 2) (NCte 2)) ]
```

- ¿Se podría hacer más eficiente ANTES de ejecutarlo?
  - Constant folding, simplification

```
p = P [ Assign “x” (NCte 17))
      , Assign “y” (Sub (NCte 59) (Var “x”)) ]
```

Manipulación simbólica (cont.)

- Expresiones sin variables

  groundNExp :: NExp -> Bool

- Simplificación y evaluación

  simplifyNExp :: NExp -> NExp
  evalGroundNExp :: NExp -> Int
  -- PRECOND: el argumento es ground

  - ¡simplify debería usar evalGroundNExp!

- Análisis simbólico del programa

  optimize :: Program -> Program

Manipulación simbólica (cont.)

- Expresiones sin variables

```
groundNExp :: NExp -> Bool
groundNExp ...
```

Manipulación simbólica (cont.)

- Simplificación y evaluación

```
simplifyNExp :: NExp -> NExp

simplifyNExp …
    -- OBS: usa evalGroundNExp


evalGroundNExp :: NExp -> Int
  -- PRECOND: el argumento es ground
evalGroundNExp …
```

Manipulación simbólica (cont.)

- Análisis simbólico del programa

  optimize :: Program -> Program

  optimize ...

Resumen

- Tipos algebraicos recursivos
  - N, BE, Listas,
- Árboles
  - Arbol a, ExpA. NExp, BExp, Comando
- Funciones sobre tipos recursivos
- Programas LIS
- Semántica (integral) de programas LIS y manipulación simbólica de programas