PROGRAMACIÓN FUNCIONAL

Tipos de Datos: Esquemas de recursión

Esquemas de Recursión

- Esquemas de trabajo sobre listas como funciones de alto orden: map, filter
- Patrón de recursión estructural sobre listas como función de alto orden: foldr
- Propiedades del patrón de recursión estructural: fusión y lluvia ácida
- Patrón de recursión estructural en otros tipos: naturales
- Patrón de recursión primitiva en listas y naturales

Esquemas de funciones

- Escriba las siguientes funciones sobre listas:

```
succl :: [ Int ] -> [ Int ]
-- suma uno a cada elemento de la lista
upperl :: [ Char ] -> [ Char ]
-- pasa a mayúsculas cada caracter de la lista
test :: [ Int ] -> [ Bool ]
-- cambia cada número por un booleano que
-- dice si el mismo es cero o no
```

- ¿Observa algo en común entre ellas?

Esquemas de funciones

- Solución:

```
succl [ ] = [ ]
succl (n:ns) = (n+1)  :  succl ns
upperl [ ] = [ ]
upperl (c:cs) = upper c  :  upperl cs
test [ ] = [ ]
test (x:xs) = (x==0)  :  test xs
```

- Sólo las partes recuadradas son distintas ¿podremos aprovechar ese hecho?

Esquemas de funciones

- Técnica de “cajas”
- Reescribimos las cajas para que no dependan de nada

```
succl [ ] = [ ]
succl (n:ns) = (\n' → n'+1) n : succl ns

upperl [ ] = [ ]
upperl (c:cs) = (\c' → upper c') c : upperl cs

test [ ] = [ ]
test (x:xs) = (\n → n ==0) x : test xs
```

Esquema de map

- La respuesta es sí:

```
map :: ??
map f [ ]  =  [ ]
map f (x:xs) = f x : map f xs
```

- Y entonces

```
succl' = map (\n' → n'+1)
upperl' = map upper
test' = map (==0)
```

- ¿Podría probar que succl' = succl? ¿Cómo?

Esquema de map

- Probamos que para toda lista finita xs,

  succl' xs = succl xs

  por inducción en la estructura de la lista.
- Caso base: xs = [ ]
  - Usar succl', map.1, y succl.1
- Caso inductivo: xs = x:xs'
  - Usar succl', map.2, succl', HI, y succl.2
- ¡Observar que no estamos contemplando el caso ⊥ ni el de listas no finitas, o con elementos ⊥!

Esquemas de funciones

- Escriba las siguientes funciones sobre listas:

```
masQueCero :: [ Int ] -> [ Int ]
-- retorna la lista que sólo contiene los números
-- mayores que cero, en el mismo orden

digitos :: [ Char ] -> [ Char ]
-- retorna los caracteres que son dígitos

noVacias :: [ [ a ] ] -> [ [ a ] ]
-- retorna sólo las listas no vacías
```

- ¿Observa algo en común entre ellas?

Esquemas de funciones

- Solución:

```
digitos [ ] = [ ]
digitos (c:cs) =
   if (isDigit c) then c : digitos cs
                  else digitos cs
noVacias [ ] = [ ]
noVacias (xs:xss) =
   if (null xs) then noVacias xss
                else xs : noVacias xss
```

- Sólo las partes recuadradas son distintas ¿podremos aprovechar ese hecho?

Esquemas de funciones

- Técnica de cajas:

```
digitos [ ] = [ ]
digitos (c:cs) =
   if (\c' → isDigit c') c   then c : digitos cs
                             else digitos cs
noVacias [ ] = [ ]
noVacias (xs:xss) =
   if (\xs' → not (null xs')) xs
    then xs : noVacias xss
    else noVacias xss
```

- ¡Observar cómo se reescriben las funciones para que se parezcan!

Esquema de filter

- La respuesta es sí:

```
filter :: ??
filter p [ ]  =  [ ]
filter p (x:xs) = if  ( p  x )  then x : filter p xs
                                else filter p xs
```

- Y entonces

```
masQueCero' = filter (>0)
digitos' = filter isDigit
noVacias' = filter (not . null)
```

- ¿Podría probar que noVacias' = noVacias?

Esquemas de funciones

- Escriba las siguientes funciones sobre listas:

```
sonCincos :: [ Int ] -> Bool
-- dice si todos los elementos son 5
```

```
all :: [ Bool ] -> Bool
-- dice si no hay ningún False en la lista
```

```
concat :: [ [ a ] ] -> [ a ]
-- hace el append de todas las listas en una
```

- ¿Observa algo en común entre ellas? ¿Qué es?

Esquemas de funciones

- ¡Todas están definidas por recursión!

```
sonCincos [ ] = True
sonCincos (n:ns) = n ==5 && sonCincos ns

all [ ] = True
all (b:bs) = b && all bs

concat [ ] = [ ]
concat (xs:xss) = xs ++ concat xss
```

- ¿En qué difieren? Sólo en el contenido de los recuadros. ¡La estructura es la misma!

Esquemas de funciones

- Aplicando la técnica de las cajas

```
sonCincos [ ] = True
sonCincos (n:ns) =
  (\x b →  x==5 && b) n (sonCincos ns)

concat [ ] = [ ]
concat (xs:xss) =
  (\ys zs →  ys ++ zs) xs (concat xss)
```

- Las cajas son más complicadas, pero la técnica es la misma

Esquema de recursión (fold)

- ¿Podemos aprovecharlo?

```
foldr :: ??
foldr f z [ ]  =  z
foldr f z (x:xs) = f x (foldr f z xs)
```

- Y entonces

```
sonCincos' = foldr check True
             where check x b = (x==5) && b
all' = foldr (&&) True
concat' = foldr (++) [ ]
```

- ¿Podría probar que concat' = concat?

Esquemas de funciones

- ¿Qué ventajas tiene trabajar con esquemas?

  Permite
  - definiciones más concisas y modulares
  - reutilizar código
  - demostrar propiedades generales
- ¿Qué requiere trabajar con esquemas?
  - Familiaridad con funciones de alto orden
  - Detección de características comunes (¡ABSTRACCIÓN!)

Propiedades de esquemas

- Demostración de propiedades
- FUSIÓN:

  si h (f x y) = g x (h y)
  entonces h . foldr f z = foldr g (h z)

- <u>Ejemplo:</u> probar que (+1) . sum = foldr (+) 1
- **<u>Dem:</u>** dado que sum = foldr (+) 0, que 0+1 = 1 y que (x+y)+1 = x+(y+1), la propiedad se concluye por fusión (con h = (+1), f = (+) y g = (+)).

Propiedades de esquemas

- <u>Propiedad:</u> probar que (n*) . sum = foldr ((+) . (n*)) 0
- **<u>Dem:</u>** dado que sum = foldr (+) 0, y que n*0 = 0, la propiedad se cumpliría por fusión, tomando
  h = (n*), f = (+) y g = ((+) . (n*))
  Faltaría ver que h (f x y) = g x (h y), o sea
  (x+y)*n = ((+) . (n*)) x (n*y)
  = (+) (n*x) (n*y)
  = n*x + n*y
  que se cumple por propiedad distributiva.

Propiedades de esquemas

- Demostración de propiedades (2)
- LLUVIA ÁCIDA (*acid rain*):

  si `g :: (A -> b -> b) -> b -> (C -> b)` para `A` y `C` fijos, entonces `foldr f z . g (:) [] = g f z`

- Debe su nombre a que elimina la estructura de datos intermedia creada por `g` (en este caso una lista, pero, en general, un árbol).

Propiedades de esquemas

- Propiedad: probar que `length . map h = length`
- **Dem:** definimos

```
g h f z = foldr (f . h) z
```

y probamos que

```
map h = g h (:) []
```

y que

```
length = g h (\_ n -> n+1) 0
```

Entonces el resultado se concluye por lluvia ácida.

Esquemas y alto orden

- ¿Cómo definir append con foldr?

```
append :: [a] -> ([a] -> [a])
append [] = \ys -> ys
append (x:xs) = \ys -> x : append xs ys
```

- Expresado así, es rutina:

```
\ys -> x : append xs ys =
                (\x' h ys -> x' : h ys) x (append xs)
```

y entonces

```
append = foldr (\x h ys -> x : h ys) id
       = foldr (\x h -> (x:) . h) id = foldr ((.) . (:)) id
```

Esquemas y alto orden

- ¿Cómo definir take con foldr?

```
take :: Int -> [a] -> [a]
take _ [] = []
take 0 (x:xs) = []
take n (x:xs) = x : take (n-1) xs
```

¡El n cambia en cada paso!

- Primero debo cambiar el orden de los argumentos

```
take' :: [a] -> (Int -> [a])
take' [] = \_ -> []
take' (x:xs) = \n -> case n of 0 -> []
                               _ -> x : take' xs (n-1)
```

Esquemas y alto orden

- ¿Cómo definir take con foldr? (Cont.)

```
take' :: [a] -> (Int -> [a])
take' = foldr g (const [])
  where g _ _ 0 = []
        g x h n = x : h (n-1)
```

y entonces

```
take :: Int -> [a] -> [a]
take = flip take'

flip f x y = f y x
```

Esquemas y alto orden

- Un ejemplo más: la función de Ackerman

  (¡con notación unaria!)

```
data One = One

ack :: Int -> Int -> Int
ack n m = u2i (ack' (i2u n) (i2u m))
            where i2u n = repeat n One
                  u2i = length

ack' :: [ One ] -> [ One ] -> [ One ]
ack' []      ys     = One : ys
ack' (x:xs) []      = ack' xs [ One ]
ack' (x:xs) (y:ys) = ack' xs (ack' (x:xs) ys)
```

Esquemas y alto orden

- La función de Ackerman (cont.)

```
ack' :: [ One ] -> [ One ] -> [ One ]
ack' []       = \ys -> One : ys
ack' (x:xs) = g
   where g []       = ack' xs [ One ]
            g (y:ys) = ack' xs (g ys)
```

- Reescribimos ack' (x:xs) = g como un foldr

```
ack' (x:xs) = foldr (\_ -> ack' xs) (ack' xs [ One ])
```

Esquemas y alto orden

- Y finalmente podemos definir ack' con foldr

```
ack' :: [ One ] -> [ One ] -> [ One ]
ack' = foldr (const g) (One :)
        where g h = foldr (const h) (h [ One ])
```

- Con esto podemos ver que la función de Ackerman termina para todo par de números naturales.

Esquemas en otros tipos

- Los esqumas de recursión también se pueden definir para otros tipos.
- Los naturales son un tipo inductivo.

```
foldNat :: (b -> b) -> b -> Nat -> b
foldNat s z 0 = z
foldNat s z n = s (foldNat s z (n-1))
```

- Los casos de la inducción son cero y el sucesor de un número, y por eso los argumentos del foldNat.

Propiedades y otros tipos

- Versiones de las propiedades de fusión y lluvia ácida valen para otros tipos también.
- FUSIÓN (para naturales):

  si h (f x) = g (h x)

  entonces h . foldNat f z = foldNat g (h z)
- LLUVIA ÁCIDA (para naturales):

  si g :: (b -> b) -> b -> (N -> b) para N fijo,

  entonces foldNat f z . g (+1) 0 = g f z

Propiedades y otros tipos

- Ejemplo:

  n + m = foldNat (+1) m n
  n * m = foldNat (+m) 0 n

- <u>Propiedad:</u> probar que (n+m)*k = n*k + m*k

- <u>**Dem:**</u>

  (n+m)* k
  = (paso 1)
  foldNat (+k) (m*k) n
  = (fusión en (+(m*k)) . (*k))
  n*k + m*k

Propiedades y otros tipos

- <u>Propiedad:</u> probar que (n+m)*k = n*k + m*k
- **<u>Dem:</u>** Paso 1)

  (n+m)* k

  = (def. (+))

  (*k) (foldNat (+1) m n)

  = (definiendo g f z = foldNat f (foldNat f z m) n)

  (*k) (g (+1) 0)

  = (def. (*) y lluvia ácida)

  (g (+k) 0)

  = (def. g y (*))

  foldNat (+k) (m*k) n

Recursión Primitiva (Listas)

- No toda función sobre listas es definible con foldr.
- Ejemplos:

```
tail :: [a] -> [a]
tail (x:xs) = xs

insert :: a -> [a] -> [a]
insert x [] = [x]
insert x (y:ys) = if x<y then (x:y:ys) else (y:insert x ys)
```

  - (**Nota:** en listas es complejo de observar. La recursión primitiva se observa mejor en árboles.)

Recursión Primitiva (Listas)

- El problema es que, además de la recursión sobre la cola, ¡utilizan la misma cola de la lista!
- Solución

```
recr :: b -> (a -> [a] -> b -> b) -> [a] -> b
recr z f [] = z
recr z f (x:xs) = f x xs (recr z f xs)
```

- Entonces

```
tail = recr (error “Lista vacía”) (\_ xs _ -> xs)
insert x = recr [x] (\y ys zs -> if x<y then (x:y:ys)
                                         else (y:zs))
```

Recursión Primitiva (Nats)

- Recursión primitiva sobre naturales

```
recNat :: b -> (Nat -> b -> b) -> Nat -> b
recNat z f 0 = z
recNat z f n = f (n-1) (recNat z f (n-1))
```

- Ejemplos (no definibles como foldNat)

```
fact = recNat 1 (\n p -> (n+1)*p)
```

-- fact n = $\prod_{i=1}^{n} i$

```
sumatoria f = recNat 0 (\x y -> f (x+1) + y)
```

-- sumatoria f n = $\sum_{i=1}^{n} f\ i$

Resumen

- Usando funciones de alto orden se pueden definir esquemas de programas
- Se obtiene modularidad, generalidad y reuso de código y propiedades sin esfuerzo adicional
- Requiere el uso fundamental de *abstracción*
- Combinando esquemas con funciones de alto orden, se obtiene un gran poder expresivo