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

Parametrización

- ¿Qué es un parámetro? Consideremos

  f x = x + $\boxed{1}$ (ó f = \x -> x + $\boxed{1}$)

  g x = x + $\boxed{17}$ (ó g = \x -> x + $\boxed{17}$)

  h x = x + $\boxed{42}$ (ó h = \x -> x + $\boxed{42}$)

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

Parametrización

- ¿Qué es un parámetro? Consideremos

f = \x -> x + $\boxed{1}$

g = \x -> x + $\boxed{17}$ \x - > x + $\boxed{\ }$

h = \x -> x + $\boxed{42}$

  - ¿Podremos aprovechar las diferencias?
    - Generar un **esquema** (código con un “agujero”)
    - ¿Cómo definir dicho “agujero”?
    - ¡Parámetros!

Parametrización

- ¿Qué es un parámetro?

```
f x  = x+ 1
g x = x+ 17
h x = x+ 42
```

Abstracción

```
(genF y) x = x+ y
```

Instanciación

```
f = genF 1
g = genF 17
h = genF 42
```

  - Sólo las partes recuadradas son distintas ¿podremos aprovechar ese hecho?
  - Técnica de los “recuadros”
  - **Parámetro**: valor que cambia en cada uso

Esquemas de funciones

- Probemos con funciones sobre listas
  - Escribir las siguientes funciones:

```
succl :: [ Int ] -> [ Int ]
-- suma uno a cada elemento de la lista
upperl :: [ Char ] -> [ Char ]
-- pasa a mayúsculas cada carácter 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 ... succl ns ...

upperl [ ]       = ...
upperl (c:cs)    = ... c ... upperl cs ...

test [ ]         = ...
test (x:xs)      = ... x ...  test xs ...
```

  - Usamos el esquema de recursión estructural sobre listas

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... pero los círculos rojos “molestan”

Esquemas de funciones

- Técnica de los "recuadros" (extendida)

```
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
```

  - Reescribimos los recuadros (azules) para que no dependan del contexto (círculos rojos)

Esquema de map

- Procedemos con la abstracción:

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

Esquema de map

- Completamos la definición

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

Esquema de map

- Completamos la definición

```
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)
```

Esquema de map

- Agregamos el tipo

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

- Y entonces

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

Esquema de map

- Agregamos el tipo

```
map :: (a-> b) -> ([a] -> [b])
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

- **Demostración**
  - Por principio de extensionalidad, probamos que se cumple succl' xs = succl xs, para todo 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

- Una vez más, con otras funciones
  - Escribir las siguientes funciones:

```
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)  =
            ... c ... digitos cs ...


noVacias [ ]        = ...
noVacias (xs:xss) =
            ... xs ... noVacias xss ...
```

  - Siempre recursión estructural

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
```

  - Otra vez, técnica de los "recuadros" extendida

Esquemas de funciones

- Solución:

```
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 el cambio en el if de noVacias para que ambas funciones se parezcan

Esquema de filter

- Procedemos con la abstracción

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

Esquema de filter

- Completamos la definición

```
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)
```

Esquema de filter

- Agregamos el tipo

```
filter :: (a->Bool) -> [a] -> [a]
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)
```

Esquema de filter

- Agregamos el tipo

```
filter :: (a->Bool) -> ([a] -> [a])
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

- Una vez más, con más complejidad
  - Escribir las siguientes funciones:

```
sonCincos :: [ Int ] -> Bool
-- dice si todos los elementos son 5
cantTotal :: [ [a] ] -> Int
-- dice cuántos elementos de tipo a hay en total
concat :: [ [a] ] -> [ a ]
-- hace el append de todas las listas en una
```

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

Esquemas de funciones

- Solución:

```
sonCincos [ ]      = ...
sonCincos (n:ns) =
             … n ... sonCincos ns ...

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

  - Recursión estructural

Esquemas de funciones

- Aplicandola técnica de las cajas

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

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

  - Los “recuadros” son más complicadas, pero la técnica 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 )
```

  - Los "recuadros" son más complicadas, pero la técnica es la misma

Esquema de recursión (fold)

- Procedemos con la abstracción

```
foldr :: ??
foldr    [ ]      = □
foldr    (x:xs) = □ x (foldr    xs)
```

Esquema de recursión (fold)

- Completamos la definición

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

Esquema de recursión (fold)

- Completamos la definición

```
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
cantTotal' = foldr ((+) . len) 0
concat' = foldr (++) [ ]
```

Esquema de recursión (fold)

- Agregamos el tipo

```
foldr :: (a->b->b->b) -> b -> [a] -> b
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
cantTotal' = foldr ((+) . len) 0
concat' = foldr (++) [ ]
```

Esquema de recursión (fold)

- Agregamos el tipo

```
foldr :: (a->b->b->b) -> b -> ([a] -> b)
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
cantTotal' = foldr ((+) . len) 0
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

- Propiedades de los esquemas
- Analicemos el ejemplo de cantTotal

```
cantTotal :: [ [a] ] -> Int
-- dice cuántos elementos de tipo a hay en total

-- canTotal [ ] = 0
-- cantTotal (xs:xss) = length xs + cantTotal xss

cantTotal = foldr (\zs n -> length zs + n) 0
-- cantTotal = foldr ((+) . length) 0
```

- ¿Hay otra forma de pensarlo?

Propiedades de esquemas

- Alternativa para cantTotal

```
cantTotal' :: [ [a] ] -> Int
cantTotal' xss = sum (map length xss)
sum = foldr (+) 0
```

- ¿Será cierto que cantTotal es igual a cantTotal'?
- Sugiere una propiedad, que para todo xs

```
foldr f z (map g xs) = foldr (f . g) z xs
```

  - O sea, procesar primero cada elemento y luego unir los resultados da lo mismo que unir los resultados procesando cada elemento al unirlo
  - ¡Demostración por inducción estructural!

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)
```

¡Observar el parámetro adicional de f!

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 (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 (Listas)

- Otras funciones que se pueden definir con recr.

```
init :: [a] -> [a]
init = recr …

maximum :: [a] -> a
maximum = recr ...
```

Recursión Primitiva (Listas)

- Otras funciones que se pueden definir con recr.

```
init :: [a] -> [a]
init = recr (error "No definida")
            (\x xs rs -> if null xs then [] else x:rs)

maximum :: [a] -> a
maximum = recr (error "No definida")
               (\x xs m -> if null xs then x
                                       else max x m)
```

Otros esquemas de listas

- En el caso de maximum o minimum, podemos identificar otro esquema:
  - el de fold de listas no vacías (foldr1)

```
maximum, minimum :: [a] -> a
maximum = foldr1 (\x m -> max x m)

minimum = foldr1 min

foldr1 :: (a->a->a) -> [a] -> a
foldr1 f (x:xs) = foldr f x xs
```

Propiedades de esquemas

- REESTRUCTURAR LAS TEÓRICAS!!!
- (seguir con esquemas de árboles)
- (luego volver a ejemplos avanzados de foldr y definición de recr)

Propiedades de esquemas

- Otras propiedades
- FUSIÓN:

  si $\quad$ h (f x y) = g x (h y)

  entonces $\;$ h . foldr f z = foldr g (h z)
- Ejemplo: probar que (+1) . sum = foldr (+) 1
- **Dem:** 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

- Propiedad: probar que (n*) . sum = foldr ((+) . (n*)) 0
- **Dem:** 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

- Propiedad: probar que (n+m)*k = n*k + m*k
- **Dem:** 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 (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