



UNCUYO
UNIVERSIDAD
NACIONAL DE CUYO

PARADIGMAS DE PROGRAMACIÓN

PARADIGMA FUNCIONAL

Dr. Pablo Vidal

Facultad de Ingeniería
Universidad Nacional de Cuyo

2022

- 1 Parcialización
- 2 Evaluación Impaciente (Eager) y Perezosa (Lazy)
- 3 Funciones de orden superior

Parcialización

Parcialización

¿Qué diferencias hay entre las siguientes funciones?

```
minimo :: (Int, Int) -> Int  
minimo (x,y) = if x>=y then x else y
```

```
minimoC :: Int -> Int -> Int  
minimoC x y = if x>=y then x else y
```

Parcialización

La función `minimoC` toma un entero y devuelve una función que dado un entero devuelve un entero. Diremos que esta función está currificada.

Curricación

- Se reduce el número de paréntesis en una expresión.
- Las funciones curricadas pueden ser aplicadas a un sólo argumento, obteniéndose así otra función que puede ser útil.

Currificación

Para que la currificación tenga sentido necesitamos que la aplicación de función asocie a izquierda.

- $\text{minimoC } 3 \ x$ es lo mismo que $(\text{minimoC } 3) \ x$
- $\text{suma } 5 \ x$ es lo mismo que $(\text{suma } 5) \ x$
- Por lo tanto: \rightarrow debe asociar a derecha
 - $\text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$ es igual a $\text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})$

Ejemplo

- $(\text{suma } 5 \ 9)$ se interpreta como la aplicación de la función suma a 5, luego la aplicación de la función resultante a 9.
- El tipo de la expresión $(\text{suma } 5 \ 9)$ es \mathbb{Z} , es decir $(\text{suma } 5 \ 9)$ es un entero y en particular su valor es 14.
- $(\text{suma } 8)$ se interpreta como la aplicación de la función suma a 8.
- El tipo de la expresión $(\text{suma } 8)$ es $\mathbb{Z} \rightarrow \mathbb{Z}$, $(\text{suma } 8)$ es una función que recibe un entero y devuelve otro entero que es el resultado de sumar el entero dado a 8.

Curricación

- Ejemplo de parcialización:

```
suma' :: Int -> (Int -> Int)
suma' x y = x+y
```

```
*Main> :type suma' 2
suma' 2 :: Int -> Int
*Main> :type suma' 2 3
suma' 2 3 :: Int
```

- Parcialización con tres argumentos
- Ejemplo de parcialización con tres argumentos:

```
mult :: Int -> (Int -> (Int -> Int))  
mult x y z = x*y*z
```

```
*ghci> :type mult 2  
mult 2 :: Int -> (Int -> Int)  
*ghci> :type mult 2 3  
mult 2 3 :: Int -> Int  
*ghci> :type mult 2 3 7  
mult 2 3 7 :: Int
```

Aplicación parcial

- Las funciones curricadas pueden aplicarse parcialmente. Por ejemplo,

```
*ghci> (suma' 2) 3
```

- Pueden definirse funciones usando aplicaciones parciales. Por ejemplo,

```
suc :: Int -> Int  
suc = suma' 1
```

- suc x es el sucesor de x. Por ejemplo, suc 2 es 3.
- Las funciones suma' y mult son curricadas.

Convenios para reducir paréntesis

- Convenio 1: Las flechas en los tipos se asocia por la derecha. Por ejemplo,

`Int -> Int -> Int -> Int`

representa a

`Int -> (Int -> (Int -> Int))`

- Convenio 2: Las aplicaciones de funciones se asocia por la izquierda. Por ejemplo,

`mult x y z`

representa a

`((mult x) y) z`

- **Nota:** Todas las funciones con múltiples argumentos se definen en forma currificada, salvo que explícitamente se diga que los argumentos tienen que ser tuplas.

Ejemplo

```
between menor mayor nro = menor <= nro && nro <= mayor
```

```
> between 5 10 7
```

```
True
```

```
(between 18 65)
```

```
debeVotar persona = (between 18 65 . edad) persona
```

Ejemplo

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

```
> (map.length) ["Hola","mundo"]
```

Ejemplo

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

```
> (map.length) ["Hola","mundo"] NUP
```

```
> (map length . (filter (\e -> (length e) < 6))) ["hola", "palabraconmuchos"]
```

Evaluación Impaciente (Eager) y Perezosa (Lazy)

Evaluación Impaciente y Perezosa

Función: $\text{doble } x = x + x$

- Ejemplo de evaluación anidada impaciente: `doble (doble 3)`

```
doble (doble 3) =
  = doble (3 + 3) [def. de doble]
  = doble 6 [def. de +]
  = 6 + 6 [def. de doble]
  = 12 [def. de +]
```

- Ejemplo de evaluación anidada perezosa:

```
doble (doble 3) =
  = (doble 3) + (doble 3) [def. de doble]
  = (3 +3) + (doble 3) [def. de doble]
  = 6 + (doble 3) [def. de +]
  = 6 + (3 + 3) [def. de doble]
  = 6 + 6 [def. de +]
  = 12 [def. de +]
```

Evaluación sobre listas

- Especificación: $\text{sum } xs$ es la suma de los elementos de xs .
- Ejemplo: $\text{sum } [2,3,7] \text{ } 12$
- Definición:

```
sum [] = 0
```

```
sum (x:xs) = x + sum xs
```

- Evaluación:

```
sum [2,3,7] =
  = 2 + sum [3,7] [def. de sum]
  = 2 + (3 + sum [7]) [def. de sum]
  = 2 + (3 + (7 + sum [])) [def. de sum]
  = 2 + (3 + (7 + 0)) [def. de sum]
  = 12 [def. de +]
```

Definición

Evaluaciones

La **Evaluación impaciente** evalúa cada expresión en el momento exacto del tiempo en que ésta es encontrada dentro del código fuente. La **Evaluación perezosa**, por el contrario, posterga la evaluación de la expresión hasta que su valor es realmente demandado por el programa en ejecución.

Ventajas:

- Codificación
- Recursión
- Rendimiento

Estrategias de evaluación

- Para los ejemplos se considera la función

```
mult :: (Int,Int) -> Int
mult (x,y) = x*y
```

- Evaluación mediante paso de parámetros por valor (o por más internos):

```
mult (1+2,2+3)
= mult (3,5) [por def. de +]
= 3*5 [por def. de mult]
= 15 [por def. de *]
```

- Evaluación mediante paso de parámetros por nombre (o por más externos):

```
mult (1+2,2+3)
= (1+2)*(3+5) [por def. de mult]
= 3*5 [por def. de +]
```

Evaluación con lambda expresiones

Se considera la función

```
mult' :: Int -> Int -> Int  
mult' x = \y -> x*y
```

Evaluación:

Funciones de orden superior

Orden Superior

- Se dice que en un lenguaje las funciones son de primera clase (o que son “objetos de primera clase”) cuando se pueden tratar como cualquier otro valor del lenguaje, es decir, cuando se pueden almacenar en variables, pasar como parámetro y devolver desde funciones, sin ningún tratamiento especial.
- El ejemplo más claro en un lenguaje popular lo encontramos en JavaScript, donde estas operaciones con funciones son muy comunes.
- Gracias al orden superior, resulta muy sencillo utilizar esquemas generales, de modo que si tenemos que implementar una serie de funciones similares, podemos hacerlo una única vez, pero parametrizando el esquema de forma adecuada para que pueda adaptarse a cada uno de los casos concretos que necesitemos. Los ejemplos del resto de la página ilustran a qué nos referimos.

Orden Superior

Una función es de orden superior si toma una función como argumento o devuelve una función como resultado.

- $(\text{dosVeces } f \ x)$ es el resultado de aplicar f a $f \ x$. Por ejemplo,

```
dosVeces (*3) 2 18
```

```
dosVeces reverse [2,5,7] [2,5,7]
```

```
dosVeces :: (a -> a) -> a -> a
```

```
dosVeces f x = f (f x)
```

- Prop: $\text{dosVeces reverse} = \text{id}$ donde id es la función identidad.

```
id :: a -> a
```

```
id x = x
```


Usos de las funciones de orden superior

- Definición de patrones de programación.
- Aplicación de una función a todos los elementos de una lista.
- Filtrado de listas por propiedades.
- Patrones de recursión sobre listas.
- Diseño de lenguajes de dominio específico:
- Lenguajes para procesamiento de mensajes.
- Analizadores sintácticos.
- Procedimientos de entrada/salida.
- Uso de las propiedades algebraicas de las funciones de orden superior para razonar sobre programas.

Funciones orden superior predefinidas

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

```
map f [] = []
```

```
map f (x:xs) = f x : map f xs
```

```
> map sqrt xs
```

Funciones orden superior predefinidas

Obviamente, podemos hacer lo mismo con cualquier otra operación. Otras funciones de orden superior muy útiles y que están predefinidas son las siguientes:

```
filter :: (a->Bool) -> [a] -> [a]
{- Devuelve sólo los elementos que cumplen la propiedad
   que se pasa como primer parámetro -}
filter p [] = []
filter p (x:xs)
  | p x = x : filter p xs
  | otherwise = filter p xs

zipWith :: (a->b->c) -> [a] -> [b] -> [c]
{- Generaliza zip aplicando cualquier función para mezclar
   los elementos de las dos listas -}
zipWith f [] ys = []
zipWith f xs [] = []
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
```

Funciones orden superior predefinidas

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

```
{- Generaliza take de modo que se cojan elementos de la lista  
hasta que aparezca el primero que no cumpla la propiedad -}
```

```
takeWhile p [] = []
```

```
takeWhile p (x:xs)
```

```
  | p x = x : takeWhile p xs
```

```
  | otherwise = []
```

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

```
{- Generaliza drop de modo que se salten todos los elementos iniciales  
de la lista hasta que aparezca uno que sí cumpla la propiedad -}
```

```
dropWhile p [] = []
```

```
dropWhile p (x:xs)
```

```
  | p x = dropWhile p xs
```

```
  | otherwise = x:xs
```

Funciones orden superior predefinidas

```
span :: (a->Bool) -> [a] -> ([a],[a])  
{- Generaliza splitAt de modo que se utilice un predicado como en  
las funciones takeWhile y dropWhile anteriores -}  
span p [] = ([],[])  
span p (x:xs)  
  | p x = (x:iz,dr)  
  | otherwise = ([],x:xs)  
  where (iz,dr) = span p xs
```

Composición y orden superior

¿Qué diferencia hay entre $(\text{even} . \text{enesimo } n)$ xs y filter even xs , más allá de que tienen objetivos diferentes?

Composición y orden superior

¿Qué diferencia hay entre $(\text{even} . \text{enesimo } n) \text{ xs}$ y filter even xs , más allá de que tienen objetivos diferentes?

- En el primer caso se construye una función adhoc a partir de la composición de dos funciones existentes: esa función recibe una lista y permite determinar si el enésimo elemento de esa lista es par.

Composición y orden superior

¿Qué diferencia hay entre `(even . enesimo n) xs` y `filter even xs`, más allá de que tienen objetivos diferentes?

- En el primer caso se construye una función adhoc a partir de la composición de dos funciones existentes: esa función recibe una lista y permite determinar si el enésimo elemento de esa lista es par.
- En el segundo caso tenemos una función como valor de primer orden. Mando la función a `filter`, y `filter` adentro llama a esa función que le paso como parámetro. Es una función de orden superior (mientras que la primera no). Lo potente es que `filter` recibe una función que ni siquiera conoce. Simplemente la usa (delega la responsabilidad a la otra función)

Definición de nuevas funciones de orden superior

Veamos un ejemplo de función de orden superior que no está predefinida pero que nos podría ser útil.

- Supongamos que estamos resolviendo una hoja de ejercicios de clase, y tenemos la implementación que se nos ha ocurrido y también tenemos la solución propuesta por el profesor.
- Queremos hacer unos tests de prueba para ver si nuestra función se comporta igual que la del profesor.

Definición de nuevas funciones de orden superior

- En general, es imposible implementar un programa que diga si dos funciones son iguales o no, pero sí podemos proporcionar un método general de testing para encontrar diferencias tras ejecutar una batería de casos de prueba.
- Consideremos la siguiente definición:

```
tester :: Eq b => (a->b) -> (a->b) -> [a] -> Bool
tester f g xs = and comparados
    where trasF = map f xs
          trasG = map g xs
          comparados = zipWith (==) trasF trasG
```

Definición de nuevas funciones de orden superior

```
tester :: Eq b => (a->b) -> (a->b) -> [a] -> Bool
tester f g xs = and comparados
    where trasF = map f xs
          trasG =map g xs
          comparados = zipWith (==) trasF trasG
```

Si la función `f` es mi solución para resolver el ejercicio 2, podemos comparar con la función `ej2` que aparece en las soluciones:

```
tester f ej2 [0,3,1,1234,9,16,4,8,100]
```

Se compararán todos los resultados de ambas funciones, comprobando si todos ellos son iguales o no. Si se retorna `False`, será porque nuestra función estaba mal, mientras que si devuelve `True` no significa necesariamente que esté bien, sólo que está bien para los casos probados.

Composición de funciones

- Definición de la composición de dos funciones:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
```

```
f . g = \x -> f (g x)
```

- Uso de composición para simplificar definiciones:

- Definiciones sin composición:

```
par n = not (impar n)
```

```
doVeces f x = f (f x )
```

```
sumaCuadradosPares ns = sum (map (^2) (filter even ns))
```

- Definiciones con composición:

```
par = not . impar
```

```
dosVeces f = f . f
```

```
sumaCuadradosPares = sum . map (^2) . filter even
```

Composición de una lista de funciones

- La función identidad:

```
id :: a -> a
```

```
id = \x -> x
```

- (composicionLista fs) es la composición de la lista de funciones fs. Por ejemplo,

```
composicionLista [(*2),(^2)] 3 18
```






```
composicionLista [(^2),(*2)] 3 36
```

```
composicionLista [(/9),(^2),(*2)] 3 4.0
```

```
composicionLista :: [a -> a] -> (a -> a)
```

```
composicionLista = foldr (.) id
```

Referencias

-  R. BIRD. , 2000, *Introducción a la programación funcional con Haskell*, Prentice Hall.
-  GRAHAM HUTTON, 2007, *Programming in Haskell*, Cambridge University Press, New York, NY, USA.
-  O'SULLIVAN, BRYAN, STEWART, DON AND GOERZEN, JOHN, 2008, *Real World Haskell*, O'Reilly.
-  B.C. RUIZ, F. GUTIÉRREZ, P. GUERRERO Y J.E. GALLARDO, 2004, *Razonando con Haskell*, Thompson.
-  S. THOMPSON , 1999, *Haskell: The Craft of Functional Programming, Second Edition*, Addison-Wesley.