



UNCUYO
UNIVERSIDAD
NACIONAL DE CUYO

PARADIGMAS DE PROGRAMACIÓN

PARADIGMA FUNCIONAL

Dr. Pablo Vidal

Facultad de Ingeniería
Universidad Nacional de Cuyo

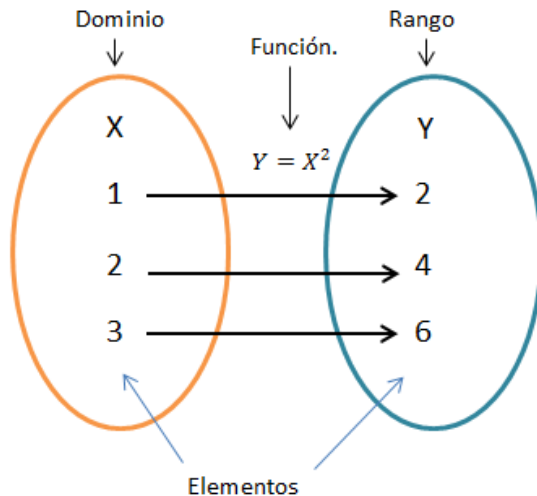
2022

Tabla de Contenidos

- 1 Introducción
 - Características
- 2 Pilares de la PF
 - Transparencia Referencial
- 3 Haskell
 - Comandos
- 4 Def. de funciones
- 5 Identificadores
- 6 IO
- 7 Listas
- 8 Tipos y Clases
- 9 Polimorfismo y sobrecarga

Introducción

Función



Existencia Y Unicidad

Una función es una relación entre dos conjuntos (dominio y codominio) mediante una relación f .

Se debe cumplir dos condiciones:

Existencia

Cada elemento tiene que estar relacionado

Unicidad

implica que cada elemento de X está relacionado con solo un elemento de Y .

Introducción

Paradigma Funcional

Es parte del paradigma declarativo basado en el uso de funciones matemáticas. Se basa en un conjunto de funciones (relaciones que cumplen las propiedades de unicidad y existencia)

Introducción

Paradigma Funcional

Está basado en conceptos que vienen de la matemática, entonces algunas cosas (p.ej. notaciones en el lenguaje) están sacadas de lo que han aprendido en Análisis I / Álgebra / Discreta.

Función

La programación funcional tiene sus raíces en el *Cálculo Lambda*, un sistema formal desarrollado en los años 1930 para investigar la definición de función, la aplicación de las funciones y la recursión. Muchos lenguajes de programación funcionales pueden ser vistos como elaboraciones del cálculo lambda.

Introducción

Paradigma Funcional

- Separa las estructuras de datos y las funciones que operan sobre ellas.
- Los programas se construyen mediante la composición de funciones, de manera que una función realiza su trabajo llamando a otras funciones cada vez más simples.

Función

La unidad de computo como característica base del paradigma.

Características

- Los bucles se modelan a través de la recursividad ya que no hay manera de incrementar o disminuir el valor de una entidad.
- Como aspecto práctico casi todos los lenguajes funcionales soportan el concepto de variable, asignación y bucle.
- Los elementos anteriores no forman parte del modelo funcional “puro”

Orígenes Históricos

- **1930s:** Alonzo Church desarrolla el lambda cálculo (teoría básica de los lenguajes funcionales).
- **1950s:** John McCarthy desarrolla el Lisp (lenguaje funcional con asignaciones).
- **1960s:** Peter Landin desarrolla ISWIN (lenguaje funcional puro).
- **1970s:** John Backus desarrolla FP (lenguaje funcional con orden superior).
- **1970s:** Robin Milner desarrolla ML (lenguaje funcional con tipos polimórficos e inferencia de tipos).
- **1980s:** David Turner desarrolla Miranda (lenguaje funcional perezoso).
- **1987:** Un comité comienza el desarrollo de Haskell.
- **2003:** El comité publica el *Haskell Report*.

Pilares de la PF

Transparencia Referencial

Los lenguajes funcionales puros tienen la propiedad de **transparencia referencial**: es posible sustituir una expresión por su valor sin que se introduzca ninguna modificación en el resultado final del programa.

Transparencia Referencial

Definición Formal

Hay transparencia referencial si al reemplazar una operación por su resultado se logra el mismo efecto. Podemos reemplazar cualquier referencia a una función por el valor que regresa sin que se altere el resultado o el comportamiento del programa.

Definición Alternativa

Hay transparencia referencial cuando al realizar una operación con los mismos valores siempre da el mismo resultado.

Si bien esta parece más fácil de entender, no es tan precisa como la primera; puede ser útil para dar los primeros pasos, pero para el final hay que terminar de entender la primera.

Transparencia Referencial

Una operación tiene transparencia referencial si:

- **Determinística**
- **Independiente**
- **Sin estado/Stateless:**
- **No produce efecto colateral**

Efecto de Lado/Colateral (Side Effect)

- Un cambio de estado sobrevive a la realización de una operación. Por ejemplo, una operación puede modificar una variable global, modificar uno de sus argumentos, escribir datos a la pantalla o a un archivo, o hacer uso de otras operaciones que tienen efecto de lado.
- Otra definición válida es: Si le sacás una foto al sistema (llamémosla F1), después realizas la operación de tu interés, y le volvés a sacar una foto al sistema (F2).
 - Si F1 y F2 son distintas \rightarrow la operación que hiciste tiene efecto de lado.

Asignación Destructiva

- Asignar destructivamente es reemplazar el valor de una variable por otro valor.
- La unificación es una asignación destructiva?? No se considera asignación (al momento de ligar no había ningún valor anterior, ¿sería más bien una inicialización?).

Ejemplo 1

Consulta no determinística

El siguiente código crea una fecha, configurada para representar el día de hoy:
`horaActual()`

Ejemplo 1

Consulta no determinística

El siguiente código crea una fecha, configurada para representar el día de hoy:
`horaActual()`

- Efecto: NO

Ejemplo 1

Consulta no determinística

El siguiente código crea una fecha, configurada para representar el día de hoy:
`horaActual()`

- Efecto: NO
- Asignación Destructiva: NO

Ejemplo 1

Consulta no determinística

El siguiente código crea una fecha, configurada para representar el día de hoy:
`horaActual()`

- Efecto: NO
- Asignación Destructiva: NO
- Transparencia Referencial: NO cumple (Con cualquiera de las 2 definiciones de transparencia referencial)

Ejemplo 2

Ejemplo 2: método con efecto

Dada la siguiente implementación de una estructura llamada ElPepe:

```
struct ElPepe {  
  var energia = 100  
  
  function volar(metros) {  
    energia = energia - (metros + 4)  
  }  
  
}
```

Analicemos el mensaje: `pepita.volar(20)`

Ejemplo 2

Ejemplo 2: método con efecto

Dada la siguiente implementación de una estructura llamada ElPepe:

```
struct ElPepe {  
  var energia = 100  
  
  function volar(metros) {  
    energia = energia - (metros + 4)  
  }  
  
}
```

Analicemos el mensaje: pepita.volar(20)

- Efecto colateral: SI, *energía* antes era 100 y luego es 76.

Ejemplo 2

Ejemplo 2: método con efecto

Dada la siguiente implementación de una estructura llamada ElPepe:

```
struct ElPepe {  
  var energia = 100  
  
  function volar(metros) {  
    energia = energia - (metros + 4)  
  }  
}
```

Analicemos el mensaje: `pepita.volar(20)`

- Efecto colateral: SI, *energía* antes era 100 y luego es 76.
- Asignación destructiva: SI

Ejemplo 2

Ejemplo 2: método con efecto

Dada la siguiente implementación de una estructura llamada ElPepe:

```
struct ElPepe {  
  var energia = 100  
  
  function volar(metros) {  
    energia = energia - (metros + 4)  
  }  
}
```

Analicemos el mensaje: `pepita.volar(20)`

- Efecto colateral: SI, *energía* antes era 100 y luego es 76.
- Asignación destructiva: SI
- Transparencia Referencial: NO, se está produciendo un efecto al disminuirse la energía de pepita. En este caso el método no retorna un valor

Ejemplo 3

Ejemplo 3: método de consulta determinística

Dada la siguiente implementación del objeto pepita:

```
method para(numero){  
    var resultado = 1  
    if(numero > 0)  
        resultado = self.para(numero - 1) * numero  
    return resultado  
}
```

Ejemplo 3

Ejemplo 3: método de consulta determinística

Dada la siguiente implementación del objeto pepita:

```
method para(numero){  
    var resultado = 1  
    if(numero > 0)  
        resultado = self.para(numero - 1) * numero  
    return resultado  
}
```

- Efecto colateral: NO

Ejemplo 3

Ejemplo 3: método de consulta determinística

Dada la siguiente implementación del objeto pepita:

```
method para(numero){  
    var resultado = 1  
    if(numero > 0)  
        resultado = self.para(numero - 1) * numero  
    return resultado  
}
```

- Efecto colateral: NO
- Asignaciones Destructivas: NO

Ejemplo 3

Ejemplo 3: método de consulta determinística

Dada la siguiente implementación del objeto pepita:

```
method para(numero){  
    var resultado = 1  
    if(numero > 0)  
        resultado = self.para(numero - 1) * numero  
    return resultado  
}
```

- Efecto colateral: NO
- Asignaciones Destructivas: NO
- Transparencia Referencial: SI

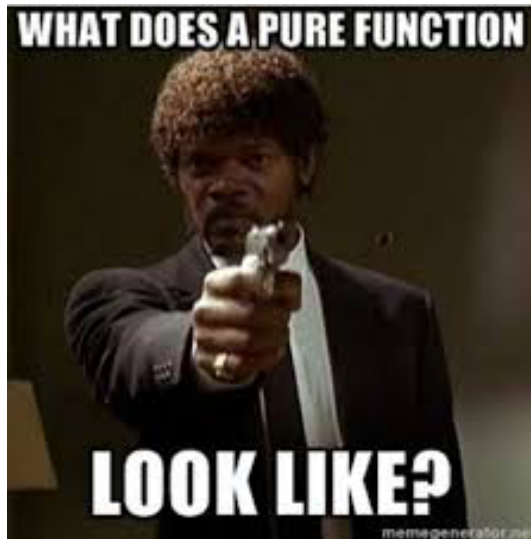
¿Por qué nos interesa pensar en estos conceptos?

- Separación entre lógica y control
- Optimización
- Testing

Tenemos transparencia referencial?

```
int a=1;  
int c;  
c=a++;
```

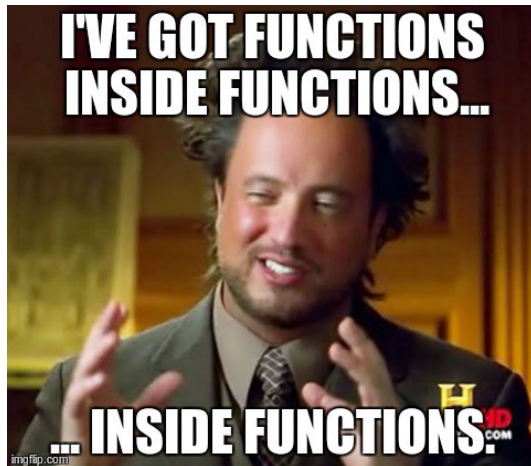
Entonces...



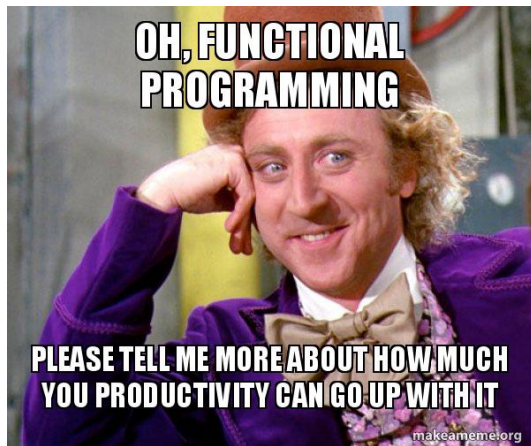
Resumiendo - Paradigma Funcional (Puro)

- Basado en el concepto (matemático) de función.
- La operación fundamental es la aplicación de una función a una serie de argumentos.
- Un programa consiste en una serie de definiciones (de funciones, datos,...)
- Las estructuras de control básicas (y generalmente únicas) son la composición y la recursión.
- No existe operación de asignación.
- Las “variables” almacenan definiciones o referencias a expresiones.

Por ello...



Pero...



Problemas

- **Interacción con el mundo exterior:** Al realizar acciones de I/O (entrada/salida) es inevitable el producir efectos laterales.
- **Eficiencia:** Si no se pueden modificar datos es necesario crear duplicados que incorporen la modificación.
- **Complejidad de programación:** Si no existe un estado externo, se debe enviar a cada función todos los datos necesarios.
- **Sistema de tipado:** Es difícil imaginar cómo incorporar un sistema de tipado estricto y/o O.O. a un enfoque funcional.

Haskell

Haskell

Síntesis diseñada por expertos de la familia ML de lenguajes de programación (1990)

- Muy influyente (C#, Python, Scala, Ruby, ...)
- Es un lenguaje funcional puro.
- Tipado Algebraico con inferencia de tipos.
- Tipado estricto y seguro.
- Funciones currificadas.
- Concordancia de Patrones.
- Evaluación perezosa/diferida.
- I/O y estilo pseudo-imperativo mediante Mónadas.

Haskell

Entorno Haskell

- www.haskell.org (GHCi, Hugs, ..)
- Se puede elegir entre modo interpretado y modo compilado.
- El modo interpretado trabaja dentro de una mónada I/O
- Contenido típico de un programa Haskell:
- Definiciones de tipos de datos
- Definiciones de funciones con su signatura de tipo
- Una función tiene el papel de punto inicial de ejecución (si se requiere interactividad se usa mónadas I/O)
- Esa función se invoca desde el intérprete.

Inicio en Haskell

Ejecutar gchi

```
GHCi, version 7.2.1: http://www.haskell.org/ghc/ :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
Prelude>
```

```
ghc --interactive
:cd dir

:set prompt ">>>"
:q or :quit
```

Cargar un archivo

Se puede cargar un archivo para poder utilizarlo de forma interactiva.

```
:load media.hs  
:l media.hs  
:r
```


Álgebra Booleana

```
ghci> True && False
False
ghci> True && True
True
ghci> False || True
True
ghci> not False
True
ghci> not (True && True)
False
```

Igualdad

```
ghci> 5 == 5
True
ghci> 1 == 0
False
ghci> 5 /= 5
False
ghci> 5 /= 4
True
ghci> "hola" == "hola"
True
```

Entonces...

Qué pasa si hacemos algo como $(5 + \text{"texto"})$ o $(5 == \text{True})$?

I/O

```
Prelude> putStrLn "Hello, Haskell"
Hello, Haskell
Prelude> putStr "No newline"
No newline
Prelude> print (5 + 4)
9
Prelude> print (1 < 2)
True
```

Def. de funciones

Función

Dado un nombre de función f
y expresiones e_1, e_2, \dots, e_k
la expresión $(f\ e_1\ e_2\ \dots\ e_k)$
representa la llamada a la función f con argumentos e_1, e_2, \dots, e_k

Función

Definir un archivo de texto que permita definir:

```
dobleF x = x + x
```

```
-- comentario
```

```
cuadrado x = x*x -- esta función calcula el  
                -- cuadrado de un número
```

Seguimos..

```
doubleVal x y = x*2 + y*2
```

```
doubleVal x y = dobleF x + dobleF y  
dobleF x = x + x
```

NOTA

Las funciones en Haskell no tienen que estar en ningún orden en particular, así que no importa si defines antes `doubleVal` y luego `dobleF` o si lo haces al revés.

funcion DO

Si se necesita múltiples acciones I/O en una expresión, se puede usar un bloque do. Las acciones se separan con punto y coma.

```
main = do { putStr "2 + 2 = " ; print (2 + 2) }
```

```
main = do putStrLn "What is 2 + 2?"  
         x <- readLn  
         if x == 4  
           then putStrLn "You're right!"  
           else putStrLn "You're wrong!"
```

Cada uno de estos pasos es una acción IO (se vera más adelante).

Def. de una función

```
hola :: String
```

```
hola = "Hola mundo"
```

```
main = do {          print( hola )}
```

Def. de una función

Una función consta de una declaración (firma) de la función y un cuerpo de la misma.

```
hola :: String -> String
```

```
hola x = "Hola " ++ x
```

```
main = do {          print( hola "Julian" )}
```

Flecha

- \rightarrow Se denomina “flecha de función” o “constructor de tipo de función”
- \rightarrow tiene un solo significado: denota una función que toma un argumento del tipo de la izquierda y devuelve un valor del tipo de la derecha

Definición

El tipo de una función f que recibe una instancia del tipo T y produce una instancia del tipo R se firma en Haskell con la siguiente signatura.

$f :: T \rightarrow R$

Identificadores

Identificadores

- Haskell provee variables inmutables (en el sentido matematico) por defecto.
- Podemos usar la def. de variable para nombrar a aquellos identificadores que usemos.

Identificadores

- Un identificador Haskell consta de una letra seguida por cero o más letras, dígitos, subrayados y comillas simples.
- `x`, `y`, `x_y`, `xs`, `ys`, `x'`, `y'`, `xs'`
- Los identificadores son case-sensitive (el uso de minúsculas o mayúsculas importa)

Identificadores

- Como en todos los lenguajes, hay “palabras reservadas”: *case, data, deriving, do, else, if, import, let, module, of, then, type, where. . .*
- La letra inicial del identificador distingue familias de identificadores: empiezan por:
 - Mayuscula los tipos y constructores de datos
 - Minuscula los nombres de función y variables

IO

I/O

```
Prelude> do { putStr "2 + 2 = " ; print (2 + 2) }  
2 + 2 = 4
```

```
Prelude> do { putStrLn "ABCDE" ; putStrLn "12345" }  
ABCDE  
12345
```

I/O - Entrada de datos

```
Prelude> do { n <- readLn ; print (n^2) }
```

```
4
```

```
16
```

Cuando se genera un programa compilado, se necesita de un punto de entrada el cual permita acceder a las funciones deseadas. Esto es una función , la cual se denomina *main*

```
main = do
  putStrLn "Hello, what's your name?"
  name <- getLine
  putStrLn ("Hey " ++ name ++ ", you rock!")
```

Ejercicio

Pedir un número y luego tener una función que incremente el valor numérico ingresado en 1. Definir la signatura de la función.

Listas

Listas

- Las listas se representan mediante la notación de corchetes.
- La lista `[1; 2; 3; 4; 5]` es de tipo `[Int]`
- La lista `[1; 2; 0;'a';'b']` es inválida porque no existe un tipo que se le pueda asignar.
- Una lista también se puede representar utilizando el constructor `:`, por ejemplo `[1; 2; 3; 4; 5]` es igual a `1 : 2 : 3 : 4 : 5 : []`
- La lista `[]` es la lista vacía.

Listas

La notación de listas aritméticas permite expresar secuencias de enteros:

`[2..10]` es `[2,3,4,5,6,7,8,9,10]`

`[1..]` es `[1,2,3,4,...]`

`[1,3..10]` es `[1,3,5,7,9]`

`[1,3..]` es `[1,3,5,7,9,...]`

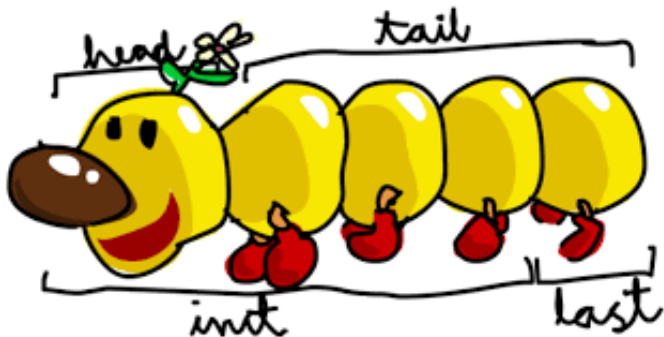
Se puede utilizar para expresar secuencias de caracteres o booleanos:

`['a'..'e']` es `"abcde"`

`[False ..]` es `[False,True]`

`['a','e'..'z']` es `"aeimquy"`

Lista



Ejercicio

Declarar y construir el cuerpo de una función que regrese la longitud de una lista:

Ejercicio

Para una lista definida de la siguiente forma

```
["maritza","celeste","nadia","maria","julia"]
```

- head
- last
- tail
- init
- length
- take
- drop
- takeWhile
- dropWhile
- reverse
- concat
- words
- unwords
- elem
- notElem

Evaluar el operador !!

Tipos y Clases

Tipos

- En Haskell, todo valor tiene asociado un tipo.
- Es un lenguaje fuertemente tipado. Acepta solamente expresiones que tengan tipo.
- Tiene inferencia de tipo. Utiliza el algoritmo de inferencia de tipos de *Hindley Milner*, que asigna a una expresión el tipo más general.
- El chequeo de tipos es **estático**, es decir, los tipos son chequeados en tiempo de compilación

Tipos simples predefinidos

- **Bool** (Valores lógicos): Sus valores son True y False.
- **Char** (Caracteres): Ejemplos: 'a', 'B', '3', '+'
- **String** (Cadena de caracteres): Ejemplos: "abc", "1 + 2 = 3"
- **Int** (Enteros de precisión fija): Enteros entre $[-2^{29} .. 2^{29}-1]$. Ejemplos: 123, -12
- **Integer** (Enteros de precisión arbitraria): Ejemplos: 1267650600228229401496703205376.
- **Float** (Reales de precisión arbitraria): Ejemplos: 1.2, -23.45, 45e-7
- **Double** (Reales de precisión doble): Ejemplos: 1.2, -23.45, 45e-7

Tipos simples predefinidos

```
ghci> :t 'a'  
'a' :: Char
```

```
ghci> :t True  
True :: Bool
```

```
ghci> :t "HOLA!"  
"HELLO!" :: [Char]
```

```
ghci> :t 4 == 5  
4 == 5 :: Bool
```

Más info

<http://hackage.haskell.org/package/base-4.12.0.0/docs/Data-Data.html>

Tipos tuplas

- Una tupla es una sucesión de elementos.
- (T_1, T_2, \dots, T_n) es el tipo de las n -tuplas cuya componente i -ésima es de tipo T_i .
- Ejemplos de tuplas:
`(False, True) :: (Bool, Bool)`
`(False, 'a', True) :: (Bool, Char, Bool)`
- Aridades:
 - La aridad de una tupla es el número de componentes.
 - La tupla de aridad 0, `()`, es la tupla vacía.
 - No están permitidas las tuplas de longitud 1

Tipos Tuplas

- Comentarios:
 - El tipo de una tupla informa sobre su longitud:
`('a', 'b') :: (Char, Char)`
`('a', 'b', 'c') :: (Char, Char, Char)`
 - El tipo de los elementos de una tupla puede ser cualquiera:

```
ghci> :t (True, 'a')  
(True, 'a') :: (Bool, Char)
```

```
((('a', 'b'), ['c', 'd'])) :: ((Char, Char), [Char])
```

Tipos funciones

- Una función es una aplicación de valores de un tipo en valores de otro tipo.
- $T1 \rightarrow T2$ es el tipo de las funciones que aplica valores del tipo $T1$ en valores del tipo $T2$.
- Ejemplos de funciones:

```
not :: Bool -> Bool
```

```
isDigit :: Char -> Bool
```

Funciones con múltiples argumentos o valores

- Ejemplo de función con múltiples argumentos:

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

- Ejemplo de función con múltiples valores:

```
deCeroA :: Int -> [Int]  
deCeroA n = [0..n]
```

- Notas:
 - En las definiciones se ha escrito la signatura de las funciones.
 - No es obligatorio escribir la signatura de las funciones.
 - Es conveniente escribir las signatura.

Tipos definidos por el usuario, Enumeraciones

Tipo Int

```
data Int = -2147483648 | -2147483647 | ... |  
-1 | 0 | 1 | 2 | ... | 2147483647
```

```
data Color = Red | Green | Blue  
data Laboral = Lu | Ma | Mi | Ju | Vi  
data MiArbol = HojaF Float | NodoI Int MiArbol MiArbol  
data ArbolBin a b = Hoja a |  
                  Nodo b (ArbolBin a b) (ArbolBin a b)
```

Constructores

```
data Bool = False | True
```

- La parte a la izquierda del = denota el tipo, que es Bool.
- La parte a la derecha son los constructores de datos.

Constructor de datos

```
data Figura = Circulo Float Float Float |  
            Rectangulo Float Float Float Float
```

```
>:t Circulo
```

```
Circulo :: Float -> Float -> Float -> Figura
```

Los constructores de datos son en realidad funciones que devuelven un valor del tipo para el que fueron definidos. Vamos a ver la declaración de tipo de estos dos constructores de datos.

Constructor de datos

```
area :: Figura -> Float
area (Circulo _ _ r) = pi * r ^ 2
area (Rectangulo x1 y1 x2 y2) =
    (abs $ x2-x1) * (abs $ y2-y1)
```

Clases básicas

- Una clase es una colección de tipos junto con ciertas operaciones sobrecargadas llamadas métodos.
- Clases básicas:
 - **Eq** tipos comparables por igualdad
 - **Ord** tipos ordenados
 - **Show** tipos mostrables
 - **Read** tipos legibles
 - **Num** tipos numéricos
 - **Integral** tipos enteros
 - **Fractional** tipos fraccionarios

La clase Eq (tipos comparables por igualdad)

- Eq contiene los tipos cuyos valores son comparables por igualdad.

- Métodos:

`(==) :: a -> a -> Bool`

`(/=) :: a -> a -> Bool`

- Instancias:

- Bool, Char, String, Int, Integer, Float y Double.
- tipos compuestos: listas y tuplas.

- Ejemplos:

`2 == 2`

`True /= True`

La clase Ord (tipos ordenados)

- Ord es la subclase de Eq de tipos cuyos valores están ordenados.

- Métodos:

`(<), (<=), (>), (>=) :: a -> a -> Bool`

`min, max :: a -> a -> a`

- Instancias:

- Bool, Char, String, Int, Integer, Float y Double.
- tipos compuestos: listas y tuplas.

- Ejemplos:

`False < True True`

`min 'a' 'b' 'a'`

`"elegante" < "elefante" False`

`[1,2,3] < [1,2] False`

`('a',2) < ('a',1) False`

`('a',2) < ('b',1) True`

La clase Show (tipos mostrables)

- Show contiene los tipos cuyos valores se pueden convertir en cadenas de caracteres.

- Método:

```
show :: a -> String
```

- Instancias:

- Bool, Char, String, Int, Integer, Float y Double.
- tipos compuestos: listas y tuplas.

- Ejemplos:

```
show False    "False"
```

```
show 'a'      "'a'"
```

```
show 123      "123"
```

```
show [1,2,3]  "[1,2,3]"
```

```
show ('a',True)  "('a',True)"
```

La clase Read (tipos legibles)

- Read contiene los tipos cuyos valores se pueden obtener a partir de cadenas de caracteres.

- Método:

```
read :: String -> a
```

- Instancias:

- Bool, Char, String, Int, Integer, Float y Double.
- tipos compuestos: listas y tuplas.

- Ejemplos:

```
read "False" :: Bool    False
read "'a'"   :: Char     'a'
read "123"   :: Int      123
read "[1,2,3]" :: [Int]   [1,2,3]
read "('a',True)" :: (Char,Bool) ('a',True)
```

La clase Num (tipos numéricos)

- Num es la subclase de Eq y Show de tipos cuyos valores son números
- Métodos:
 $(+)$, $(*)$, $(-)$ `:: a -> a -> a`
`negate`, `abs`, `signum` `:: a -> a`
- Instancias: Int, Integer, Float y Double.
- Ejemplos:

```
2+3    5
2.3+4.2 6.5
negate 2.7    -2.7
abs (-5)     5
signum (-5)  -1
```

La clase Integral (tipos enteros)

- Integral es la subclase de Num cuyo tipos tienen valores enteros.

- Métodos:

```
div :: a -> a -> a
```

```
mod :: a -> a -> a
```

- Instancias: Int, Integer.

- Ejemplos:

```
11 div 4    2
```

```
11 mod 4    3
```

La clase Fractional (tipos fraccionarios)

- Fractional es la subclase de Num cuyo tipos tienen valores no son enteros.
- Métodos:
 - `(/) :: a -> a -> a`
 - `recip :: a -> a`
- Instancias: Float y Double.
- Ejemplos:

```
7.0 / 2.0    3.5
```

```
recip 0.2    5.0
```

deriving

- La cláusula **deriving** produce de forma implícita una declaración de instancia X
- Las instancias de Ord, Enum, Read, Show, entre otras pueden ser generadas por la cláusula *deriving*
- Tras *deriving*, más de una clase puede ser especificada, en cuyo caso la lista de nombres deberá ir entre paréntesis y separados por comas

Ejemplos - deriving

```
data Figura = Circulo Float Float Float |  
            Rectangulo Float Float Float Float deriving (Show)
```

```
ghci> Circle 10 20 5  
Circle 10.0 20.0 5.0
```

```
ghci> Rectangle 50 230 60 90  
Rectangle 50.0 230.0 60.0 90.0
```

Ejemplos - deriving

```
data Dia = Domingo | Lunes | Martes | Miercoles  
         | Jueves | Viernes | Sabado deriving (Enum)
```

```
-- / Jugador de ajedrez  
data Jugador = Blanco | Negro deriving (Eq)
```

```
-- / Piezas del juego ajedrez  
data Pieza = Rey | Reina | Alfil |  
            Caballo | Torre | Peon deriving (Eq)
```

Polimorfismo y sobrecarga

Tipos polimórficos

- Un tipo es polimórfico (“tiene muchas formas”) si contiene una variable de tipo.
- Una función es polimórfica si su tipo es polimórfico.
- La función `length` es polimórfica:
- Comprobación:

```
Prelude> :type length  
length :: [a] -> Int
```
- Significa que para cualquier tipo a , `length` toma una lista de elementos de tipo a y devuelve un entero.

Tipos polimórficos

- a es una variable de tipos.
- Las variables de tipos tienen que empezar por minúscula.
- Ejemplos:
 - `length [1, 4, 7, 1]` 4
 - `length ["Lunes", "Martes", "Jueves"]` 3
 - `length [reverse, tail]` 2

Ejemplos de funciones polimórficas

```
fst :: (a, b) -> a
fst (1, 'x')      1
fst (True, "Hoy") True
```

```
head :: [a] -> a
head [2,1,4]      2
head ['b','c']    'b'
```

Ejemplos de funciones polimórficas

```
take :: Int -> [a] -> [a]
take 3 [3,5,7,9,4]    [3,5,7]
take 2 ['l','o','l','a'] "lo"
take 2 "lola"         "lo"

zip :: [a] -> [b] -> [(a, b)]
zip [3,5] "lo"        [(3,'l'),(5,'o')]
```

Tipos sobrecargados

- Un tipo está sobrecargado si contiene una **restricción de clases**
 - Un tipo está sobrecargado si contiene una restricción de clases.
 - Una función está sobrecargada si su tipo está sobrecargado.

- La función `sum` está sobrecargada:

- Comprobación:

```
ghci> :type sum
sum :: (Num a) => [a] -> a
```

- Significa que para cualquier tipo numérico `a`, `sum` toma una lista de elementos de tipo `a` y devuelve un valor de tipo `a`.
- `Num a` es una restricción de clases.

Tipos sobrecargados

- Las restricciones de clases son expresiones de la forma $C\ a$, donde C es el nombre de una clase y a es una variable de tipo.
- Ejemplos:

```
sum [2, 3, 5]    10
```

```
sum [2.1, 3.23, 5.345]    10.675
```

Ejemplos de tipos sobrecargados

Ejemplos de funciones sobrecargadas:

```
(-) :: (Num a) => a -> a -> a
```

```
(*) :: (Num a) => a -> a -> a
```

```
negate :: (Num a) => a -> a
```

```
abs :: (Num a) => a -> a
```






```
signum :: (Num a) => a -> a
```

Ejemplos de números sobrecargados:

```
5 :: (Num t) => t
```

```
5.2
```

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.