Notas para final de PLP

Manuel Panichelli

February 2, 2022

Chapter 1

Paradigma funcional

1.1 Haskell

Def. 1.1 (Paradigma)**.** Un **paradigma** es una forma de pensamiento.

Def. 1.2 (Lenguaje de programación)**.** Un **lenguaje de programación** es el lenguaje que usamos para comunicar lo que queremos que haga una computadora.

Usamos un lenguaje para describir los computos que lleva a cabo la computadora.

Es **computacionalmente completo** si puede expresar todas las funciones computables. Hay DSLs (*domain specific languages*) que no pueden expresar todo lo computable.

Def. 1.3 (Paradigma de lenguaje de programación)**.** Lo entendemos como un *estilo* de programación, que tiene que ver con los estilos de las soluciones. Está vinculado con lo que es para uno un modelo de cómputo.

Lo que vemos antes de la materia es el imperativo: a partir de un estado inicial llegar a un estado final. Programamos con secuencias de instrucciones para cambiar el estado.

1.2 Programación funcional

Definiciones:

- **Programa y modelo de cómputo**: Programar es definir funciones, y ejecutar es evaluar expresiones.
- **Programa**: Es un conjunto de ecuaciones. Por ej. `doble x = x + x`
- **Expresiones**: El significado de una expresión es su valor (si es que está definido). El valor de una expresión depende solo del valor de sus subexpresiones. Evaluar o reducir una expresion es obtener su valor (por ej.

`doble 2` $\rightsquigarrow$ `4`) No toda expresion denota un valor, por ejemplo `doble true`.

- **Tipos**: El universo de valores está particionado en colecciones denominadas *tipos*, que tienen operaciones asociadas.

Haskell es **fuertemente tipado**. Toda expresion bien formada tiene un tipo, que depende del tipo de sus subexpresiones. Si no puede asignarse un tipo a una expresión, no se la considera bien formada.

```
1           :: Int
'a'         :: Char
1.2         :: Float
True        :: Bool
[1, 2, 3]   :: [Int]
(1, True)   :: (Int, Bool)
succ        :: Int -> Int
```

Definiciones de funciones:

```
-- Definición
doble :: Int -> Int
doble x = x + x

-- Guardas
signo :: Int -> Bool
signo n | n >= 0    = True
        | otherwise = False

-- Definiciones locales
f (x, y) = g x + y
    where g z = z + 2

-- Expresiones lambda
\x -> x + 1
```

Tipos polimórficos

```
id x = x
id :: a -> a
-- x es de tipo a, que eventualmente se va a instanciar a algún tipo
```

Clases de tipos: Son como interfaces, que definen un conjunto de operaciones.

```
maximo :: Ord a => a -> a -> a
maximo x y | x > y = x
maximo _ y = y
-- Ord: (<), (<=), (>=), (>), max, min, compare
```

Tipos algebráicos

```
data Figura = Circulo Float | Rectangulo Float Float
deriving Eq -- deriva la igualdad nativa

-- (Circulo 1) == (Circulo 1)
```

Estas cosas nos permiten hacer funciones genéricas.

Funciones de alto orden: las funciones son first class citizens, se pueden pasar como parámetro.

1.2.1 Currificación

Es un mecanismo que permite reemplazar argumentos estructurados por una secuencia de argumentos "simples". Ventajas:

- Evaluación parcial: `succ = suma 1`
- Evita escribir paréntesis (asumiendo que la aplicación asocia a izquierda). `suma 1 2 = ((suma 1) 2)`

curry y uncurry

En criollo: una equivalencia entre una func con muchos parametros (una tupla) y una funcion equivalente que va tomando de a uno y devuelve funciones.

```
curry :: ((a, b) -> c) -> (a -> (b -> c))
curry f a b = f (a, b)

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

curry suma' 1 2 = suma' (1, 2)
curry suma' :: (Int -> (Int -> Int))

uncurry :: (a -> b -> c) -> ((a -> b) -> c)
uncurry f (a, b) = f a b
```

1.2.2 Pattern matching

Una forma copada de definir funciones. Es un mecanismo para comparar un valor con un patrón. Si la comparación tiene éxito se puede deconstruir un valor en sus partes.

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

area :: Figura -> Float
area (Circulo radio) = pi * radio ^ 2
area (Rectangulo l1 l2) = l1 * l2
```

El patrón está formado por el constructor y las variables. Los casos se evalúan en el orden en el que están escritos.

```
esCuadrado :: Figura -> Bool
-- No vale esto?
-- esCuadrado (Rectangulo x y) = (x == y)
esCuadrado (Rectangulo x y) | (x==y) = True
esCuadrado _ = False
```

También se pueden definir funciones parciales (que no estén definidas para todo el dominio).

1.2.3 Tipos recursivos

La definición de un tipo puede tener uno o más parámetros del tipo

```
data Natural = Zero | Succ Natural

Zero :: Natural                     -- 0
succ Zero :: Natural                -- 1
succ (succ (succ Zero)) :: Natural  -- 2

dameNumero :: Natural -> Int
dameNumero Zero = 0
dameNumero (Succ n) = dameNumero n + 1
```

1.2.4 Listas

Tipo algebráico paramétrico recursivo con dos constructores:

```
[] :: [a]               -- lista vacia
(:) :: a -> [a] -> [a]  -- constructor infijo

-- Ejemplo
--   1 : [2, 3] = [1, 2, 3]
```

Pattern matching

```
vacia :: [a] -> Bool
vacia [] = True
vacia _ = False

long :: [a] -> Int
long [] = 0
long x:xs = 1 + long xs
```