Notas para final de PLP

Manuel Panichelli

February 9, 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.

1.1.1 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` ⇝ 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.

1.1.2 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
```

1.1.3 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).

1.1.4 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
```

1.1.5 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
```

1.1.6 No terminación y orden de evaluación

```haskell
-- No terminación
inf1 :: [Int]
inf1 = 1 : inf1

-- Evaluación no estricta
const :: a -> b -> a
const x y = x

-- const 42 inf1 -> 42 (pero depende del mecanismo de reducción del
-- lenguaje)
```

1.1.7 Evaluación lazy

el modelo de cómputo de haskell es la **reducción**. Se reemplaza un *redex* por otro usando las ecuaciones orientadas. Un redex (reducible expression) es una sub-expresión que no está en forma normal (irreducible).

Un redex debe ser una **instancia** del lado izquierdo de alguna ecuación y será reemplazado por el lado derecho con las variables correspondientes ligadas. El resto de la expresión no cambia.

Haskell hace esto hasta llegar a una forma normal, un valor irreducible.

`const x y = x`. `const x y` es un redex, y lo reduzco a `x`.

Y cómo selecciono una redex? **Orden normal** (lazy). Se selecciona el redex más externo para el que se pueda conocer que ecuación del programa utilizar. En general, primero las funciones más externas y luego los argumentos, solo de ser necesarios.

Modo aplicativo: reduce primero todos los argumentos. Se hace en otros lenguajes como c.

1.1.8 Esquemas de recursion

Formas de recursion comunes que uno puede aprovechar usando funciones de alto orden.

Map

```haskell
-- tal que dobleL xs es la lista que contiene el doble de cada elemento en xs
dobleL :: [Float] -> [Float]
dobleL [] = []
dobleL (x:xs) = 2*x : dobleL xs

-- tal que la lista esParL xs indica si el correspondiente elemento en xs es par
-- o no
esParL :: [Int] -> [Bool]
esParL [] = []
esParL (x:xs) = (even x) : esParL xs
```

```haskell
-- tal que longL xs es la lista que contiene las longitudes de las listas en xs
longL :: [[a]] -> [Int]
longL [] = []
longL (x:xs) = (length x) : longL xs

-- esquema recursivo de map:
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map g (x:xs) = g x : map g xs

-- Con eso, se pueden reescribir como
dobleL = map ((*) 2)
esParL = map even
longL = map length
```

Filter

```haskell
-- tal que negativos xs contiene los elementos negativos de xs
negativos :: [Float] -> [Float]
negativos [] = []
negativos (x:xs)
    | x < 0 = x : (negativos xs)
    | otherwise = negativos xs

-- tal que la lista noVacias xs contiene las listas no vacias de xs
noVacias :: [[a]] -> [[a]]
noVacias [] = []
noVacias (l:ls)
    | (length l > 0) = l : (noVacias ls)
    | otherwise = noVacias ls

-- esquema recursivo:
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs) = if (p x) then x : (filter p xs)
                  else (filter p xs)

-- luego quedan
negativos = filter (\x -> x < 0)
noVacias = filter (\l -> length l != 0)
noVacias = filter ((> 0) . length) -- f o g = f(g(x))
```

1.1.9 Transparencia referencial

El valor de una expresion en funcional depende solo de sus subexpresiones. Esto a diferencia de imperativo que depende del estado.

Si dos expresiones son iguales, denotan el mismo valor bajo el mismo contexto.

1.1.10 Folds

`foldr`

```haskell
-- Funciones sobre listas

-- sumaL: suma de todos los valores de una lista de enteros
sumaL :: [Int] -> Int
sumaL [] = 0
sumaL (x:xs) = x + (sumaL xs)

-- concat: la concatenación de todos los elementos de una lista de listas
concat :: [[a]] -> [a]
concat [] = []
concat (l:ls) = l ++ (concat ls)

-- reverso: el reverso de una lista
reverso :: [a] -> [a]
reverso [] = []
reverso (x:xs) = (reverso xs) ++ [x]

-- Esquema de recursión
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ z [] = z
foldr f z (x:xs) = f x (foldr f z xs)

-- Luego, con esto
sumaL = foldr (+) 0
concat = foldr (++) []
reverso = foldr (\x rec -> rec ++ [x]) []
reverso = foldr ( (flip (++)) . (:[])) []

-- Hasta podemos definir map y filter. El fold es más general que el map y
-- filter
map f = foldr (\x rec -> f x : rec) []
map f = foldr ((:) . f) []

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

```haskell
filter p = foldr (\x rec -> if p x then x : rec else rec) []

-- Longitud y suma con una sola pasada sobre la lista
sumaLong :: [Int] -> (Int, Int)
sumaLong = foldr (\x (rl, rn) -> (rl + 1, rn + x)) (0, 0)
```

recr

```haskell
-- dropWhile
dropWhile :: (a -> Bool) -> [a] -> [a]
dropWhile _ [] = []
dropWhile p (x:xs) = if p x then dropWhile p xs else x:xs

-- ejemplo
dropWhile even [2, 4, 1, 6] = [1, 6]

-- drop while cuando se termina de cumplir devuelvo todo lo que viene "a la
-- derecha", pero cuando hago fold, lo que está a la derecha ya pasó por la
-- recursión.

dw p = first $ (foldr (\x (r1, r2)
    -> (if p x then r1 else x: r2, x: r2 ) ), ([], []))

-- Otro esquema más poderoso
g :: [a] -> b
g [] = z
g (x:xs) = f x xs (g xs)

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

dropWhile p = recr [] (\x xs rec -> if p x then rec else x:xs)

-- foldr en terminos de recr?
foldr f z = recr z (\x xs rec -> f x rec)

-- recr en términos de foldr?
recr z f = first $
    foldr
        (\x (rs1, rs2) -> (f x rs2 rs1, x:rs2))
        (z, [])
```

foldl

```haskell
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl _ z [] = z
foldl f z (x:xs) = foldl f (f z x) xs
```

foldr = fold a la derecha, y foldl = fold a la izquierda

```haskell
reverse = foldl (\c x -> x:c) []
reverse = foldl (flip (:)) []
```

Y uno en términos del otro? *Me falta repasar esto porque estaba matado, min 2:35:10 del video*

Fold sobre estructuras algebrácias

```haskell
-- Arbol binario
data Arbol a = Hoja a | Nodo a (Arbol a) (Arbol a)

-- Por ej.

Nodo 1 (Hoja 2) (Hoja 3)

-- Es
--
--   1
--  / \
-- 2   3

-- Y sobre ella podemos querer operaciones, como map

mapA :: (a -> b) -> Arbol a -> Arbol b
mapA f (Hoja x) = Hoja f x
mapA f (Nodo x izq der) = Nodo (f x) (mapA f izq) (mapA f der)

-- Y también podemos hacer un fold

foldA :: (a -> b) -> (a -> b -> b -> b) -> Arbol a -> b
foldA f g (Hoja x) = f x
foldA f g (Nodo x izq der) = g x (foldA f g izq) (foldA f g der)

sumaA = foldA id (\x rizq rder -> x + rizq + rder)
contarHojas = foldA 1 (\x rizq rder -> rizq + rder)

-- Arboles generales
data AG = NodoAG a [AG a]
```

```
mapAG :: (a -> b) -> AG a -> AG b
mapAG f (Nodo AG a as) -> NodoAG (f a) (map (mapAG f) as)

foldAG :: (a -> [b] -> b) -> AG a -> b
foldAG f (NodoAG a as) = f a (map (foldAG f) as)
```

Aplicar un fold con un constructor es la identidad.

1.2 Cálculo Lambda Tipado

Es el formalismo que está detrás de la programación funcional. Es un modelo de cómputo basado en **funciones**, introducido por Alonzo Church en 1934. Es computacionalmente completo (turing completo) y nosotros vamos a estudiar la variante tipada (Church, 1941.)

La máquina de turing es más de estado, y este con reducción de expresiones.

Def. 1.4 (Tipos)**.** Las **expresiones de tipos** (o tipos) de λ^b (lambda cálculo b de booleano) son

$$\sigma, \tau ::= Bool \mid \sigma \to \tau$$

Informalmente,

- $Bool$ es el tipo de los booleanos, y
- $\sigma \to \tau$ es el tipo de las funciones de tipo σ en tipo τ.

Ejemplo. Ejemplos:

- $Bool \to Bool$
- $Bool \to Bool \to Bool$

Def. 1.5 (Terminos)**.** Los términos se escriben con las siguientes reglas de sintaxis.

Sea χ un conjunto infinito enumerable de variables, y $x \in \chi$. Los **términos** de λ^b están dados por,

$$\begin{aligned} M, N, P, Q ::= \; & x \\ \mid \; & true \\ \mid \; & false \\ \mid \; & if\ M\ then\ P\ else\ Q \\ \mid \; & \lambda x : \sigma . M \\ \mid \; & M\ N \end{aligned}$$

Ejemplo. Ejemplos:

- $\lambda x : Bool.x$ ✓
- $\lambda x : Bool.\ if\ x\ then\ false\ else\ true$ ✓
- $\lambda f : Bool \to Bool \to Bool.\lambda x : Bool.f\ x$ ✓
- $(\lambda f : Bool \to Bool.f\ true)(\lambda y : Bool.y)$ ✓
- $true\ (\lambda x : Bool.x)$ ✓
- $x\ y$ ✓
- $\lambda x : true$ ✗

1.2.1 Sistema de tipado

Es un sistema formal de deducción o derivación que utiliza axiomas y reglas de inferencia para caracterizar un subconjunto de los términos llamados **tipados**. Nos permite quedarnos con algunos y rechazar otros términos en base a lo que consideremos correcto.

Definimos una **relación de tipado** en base a reglas de inferencia.

- Los **axiomas de tipado** establecen que ciertos **juicios de tipado** son derivables.
- Las **reglas de tipado** establecen que ciertos **juicios de tipado** son derivables siempre y cuando ceiertos otros lo sean.

Def. 1.6 (Variables libres)**.** Una variable puede ocurrir **libre** o ligada en un término. Decimos que x ocurre **libre** si no se encuentra bajo el alcance de una ocurrencia de λx. Caso contrario ocurre ligada.

Ejemplos:

- $\lambda x : Bool.\ if\ \underbrace{x}_{\text{ligada}}\ then\ true\ else\ false$
- $\lambda x : Bool.\lambda y : Bool.\ if\ true\ then\ \underbrace{x}_{\text{ligada}}\ else\ \underbrace{y}_{\text{ligada}}$
- $\lambda x : Bool.\ if\ \underbrace{x}_{\text{ligada}}\ then\ true\ else\ \underbrace{y}_{\text{libre}}$
- $(\lambda x : Bool.\ if\ \underbrace{x}_{\text{ligada}}\ then\ true\ else\ false)\ \underbrace{x}_{\text{libre}}$

La definición formal es a partir de cada término del lambda cálculo por pattern matching. FV = Free Variable

$$FV(x) \stackrel{\text{def}}{=} \{x\}$$
$$FV(true) = FV(false) \stackrel{\text{def}}{=} \emptyset$$
$$FV(\ if\ M\ then\ P\ else\ Q) \stackrel{\text{def}}{=} FV(M) \cup FV(P) \cup FV(Q)$$
$$FV(M\ N) \stackrel{\text{def}}{=} FV(M) \cup FV(N)$$
$$FV(\lambda x : \sigma . M) \stackrel{\text{def}}{=} FV(M) \setminus \{x\}$$

Def. 1.7 (Juicio de tipado)**.** Un **juicio de tipado** es una expresión de la forma $\Gamma \rhd M : \sigma$, que se lee *"El término M tiene el tipo σ asumiendo el contexto de tipado Γ"*.

Un **contexto de tipado** es un conjunto de pares $x_i : \sigma_i$, notado $\{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ donde los x_i son distintos. Usamos las letras $\Gamma, \Delta, \ldots$ para contextos de tipado.

A las variables se les anota un tipo. Uno pone las asunciones que tiene sobre el tipo de algunas variables, como $x : Bool$.

Def. 1.8 (Axiomas de tipado de λ^b)**.** Son guiadas por sintaxis al igual que las variables libres

$$\frac{}{\Gamma \rhd true : Bool}\text{(T-True)} \qquad \frac{}{\Gamma \rhd false : Bool}\text{(T-False)}$$

(para cualquier contexto de tipado Γ)

$$\frac{x : \sigma \in \Gamma}{\Gamma \rhd x : \sigma}\text{(T-Var)}$$

$$\frac{\Gamma \rhd M : Bool \quad \Gamma \rhd P : \sigma \quad \Gamma \rhd Q : \sigma}{\Gamma \rhd\ if\ M\ then\ P\ else\ Q : \sigma}\text{(T-If)}$$

P y Q tienen que tener el mismo tipo porque queremos que la expresion siempre tipe a lo mismo.

$$\frac{\Gamma, x : \sigma \rhd M : \tau}{\Gamma \rhd \lambda x : \sigma . M : \sigma \to \tau}\text{(T-Abs)} \qquad \frac{\Gamma \rhd M : \sigma \to \tau \quad \Gamma \rhd N : \sigma}{\Gamma \rhd M\ N : \tau}\text{(T-App)}$$

Si $\Gamma \rhd M : \sigma$ puede derivarse usando los axiomas y reglas de tipado, decimos que es **derivable**, y decimos que M es **tipable** si el juicio de tipado $\Gamma \rhd M : \sigma$ puede derivarse para algún Γ y σ.

Ejemplos:

- $\lambda x : Bool . x : Bool \to Bool$

$$\dfrac{\dfrac{\dfrac{\checkmark}{x : Bool \in \Gamma'}}{\Gamma' = \Gamma \cap \{x : Bool\} \rhd x : Bool}\text{(T-Var)}}{\Gamma \rhd \lambda x : Bool.x : Bool \to Bool}\text{(T-Abs)}$$

- $\lambda x : Bool.\ if\ x\ then\ false\ else\ true$
- $\lambda f : Bool \to Bool \to Bool.\lambda x : Bool.f\ x$
- $(\lambda f : Bool \to Bool.f\ true)(\lambda y : Bool.y)$
- $true\ (\lambda x : Bool.x)$. No va a tipar nunca, porque $true : Bool$ y para $T-App$ necesitamos que sea $\sigma \to \tau$.
- $x\ y$. Para usar T-App, x por sintaxis solo aplica a T-Var. La única forma de que pueda aplicar x con y, x tiene que ser tipo flecha. Pero es una variable, entonces solo funcionaría si tenemos como asunción de tipo de x como función en Γ. Sin eso no se puede tipar.

1.2.2 Resultados básicos

Se pueden probar por inducción en la longitud de las reglas

Prop. 1.1 (Unicidad de tipos)**.** Si $\Gamma \rhd M : \sigma$ y $\Gamma \rhd M : \tau$ son derivables, entonces $\sigma = \tau$.

Si una expresión tiene un tipo, ese tipo es único.

Prop. 1.2 (Weakening + Strengthening)**.** Si $\Gamma \rhd M : \sigma$ es derivable y $\Gamma \cap \Gamma'$ contiene a todas las variables libres de M, entonces $\Gamma' \rhd M : \sigma$.

Puedo agrandar o achicar el contexto de tipo siempre y cuando contenga las mismas variables libres.

1.2.3 Semántica

Habiendo definido la sintaxis de λ^b, nos interesa formular como se **evalúan** o **ejecutan** los términos. Hay varias maneras de definir **rigurosamente** la semántica de un lenguaje de programación, nosotros vamos a definir una **semántica operacional**.

- Denotacional. Darle una *denotación* a cada símbolo del lenguaje, qué denota matemáticamente. Y uno define la semántica en términos de como las funciones van denotando cosas, con recursión o puntos fijos.
- Axiomática. Cuando cursamos algo1, definimos pre y pos condiciones. Predicaque definen el significado de una operación. Las triplas de hoare y esas cosas.
- **Operacional** consiste en

- interpretar a los **términos como estados** de una máquina abstracta, y
- definir una **función de transición** que indica dado un estado cuál es el siguiente.

El **significado** de un término M es el estado final que alcanza la máquina al comenzar con M como estado inicial. Hay dos formas de definir semántica operacional,

- **Small-step**: la función de transición describe un paso de computación. Esta vamos a hacer nosotros.
- **Big-step** (o **Natural Semantics**): la función de transición, en un paso, evalúa el término a su resultado.

Def. 1.9 (Juicios)**.** La formulación se hace a través de **juicios de evaluación** $M \to N$, que se leen *"el término M reduce, en un paso, al término N"*.

El significado de un juicio de evaluación se establece a través de:

- **Axiomas de evaluación**, que establecen que ciertos juicios de evaluación son derivables.
- **Reglas de evaluación**, que establecen que ciertos juicios de evaluación son derivables siempre y cuando ciertos otros lo sean

(análogo a axiomas y reglas de tipado)

Semántica operacional small-step de λ^b

Además de introducir la función de transición es necesario introducir también los **valores**, los posibles resultados de evaluación de términos bien-tipados (derivables) y cerrados (sin variables libres).

Valores

$$V ::= true \mid false$$

todo término bien-tipado y cerrado de tipo Bool evalúa, en cero (directamente) o más pasos, a true o false. Se puede demostrar formalmente.

Juicio de evaluación en un paso:

$$\frac{}{if\ true\ then\ M_2\ else\ M_3 \to M_2}\text{(E-IfTrue)}$$

$$\frac{}{if\ false\ then\ M_2\ else\ M_3 \to M_3}\text{(E-IfFalse)}$$

$$\frac{M_1 \to M_1'}{if\ M_1\ then\ M_2\ else\ M_3 \to\ if\ M_1'\ then\ M_2\ else\ M_3}\text{(E-If)}$$