

Abordaje Funcional a EDSLs

Alberto Pardo Marcos Viera

Instituto de Computación, Facultad de Ingeniería
Universidad de la República, Uruguay

ECI 2024

Programación Funcional y EDSLs

¿Por qué Programación Funcional?

Larga tradición en la comunidad de Programación Funcional en manipulación de términos.

¿Por qué Programación Funcional?

Larga tradición en la comunidad de Programación Funcional en manipulación de términos.

Principales motivos:

- Sintaxis simple
- Nivel de Abstracción
- Tipos Algebraicos, pattern matching y recursión
- Funciones de alto orden
- Polimorfismo
- Pureza, simplicidad de razonar
- Evaluación Perezosa

¿Por qué Programación Funcional?

Larga tradición en la comunidad de Programación Funcional en manipulación de términos.

Principales motivos:

- Sintaxis simple
- Nivel de Abstracción
- **Tipos Algebraicos**, pattern matching y recursión
- **Funciones de alto orden**
- Polimorfismo
- Pureza, simplicidad de razonar
- Evaluación Perezosa

Tipos Algebraicos

Notación de GADTs:

```
data Expr where  
  Val :: Int → Expr  
  Add :: Expr → Expr → Expr
```

Tipos Algebraicos

Notación de GADTs:

```
data Expr where  
  Val :: Int → Expr  
  Add :: Expr → Expr → Expr
```

Notación clásica:

```
data Expr = Val Int | Add Expr Expr
```

Tipos Algebraicos

Notación de GADTs:

```
data Expr where  
  Val :: Int → Expr  
  Add :: Expr → Expr → Expr
```

Notación clásica:

```
data Expr = Val Int | Add Expr Expr
```

En general:

```
data T a1 ... am = C1 t11 ... t1k1  
    ...  
    | Cn tn1 ... tnkn
```

donde las variables a_i pueden ser usadas en la definición de los constructores.

Tipos Algebraicos (2)

Una manera simple de definir estructuras arborescentes:

```
data Tree a = Leaf a | Fork (Tree a) (Tree a)
```

Tipos Algebraicos (2)

Una manera simple de definir estructuras arborescentes:

```
data Tree a = Leaf a | Fork (Tree a) (Tree a)
```

```
data Tree a where  
  Leaf :: a → Tree a  
  Fork :: Tree a → Tree a → Tree a
```

Tipos Algebraicos (2)

Una manera simple de definir estructuras arborescentes:

```
data Tree a = Leaf a | Fork (Tree a) (Tree a)
```

```
data Tree a where  
  Leaf :: a → Tree a  
  Fork :: Tree a → Tree a → Tree a
```

Los términos de un lenguaje son estructuras arborescentes

Tipos Algebraicos - Constructores

Volviendo al tipo de las expresiones en un **deep embedding**:

```
data Expr where  
  Val  :: Int → Expr  
  Add  :: Expr → Expr → Expr
```

Tipos Algebraicos - Constructores

Volviendo al tipo de las expresiones en un [deep embedding](#):

```
data Expr where  
  Val  :: Int → Expr  
  Add  :: Expr → Expr → Expr
```

Los constructores se introducen al definir el tipo.

Tipos Algebraicos - Constructores

Volviendo al tipo de las expresiones en un **deep embedding**:

```
data Expr where  
  Val  :: Int → Expr  
  Add  :: Expr → Expr → Expr
```

Los constructores se introducen al definir el tipo.

También puedo definir **smart constructors**

```
val  = Val  
add  = Add
```

Tipos Algebraicos - Constructores

Volviendo al tipo de las expresiones en un **deep embedding**:

```
data Expr where  
  Val  :: Int → Expr  
  Add  :: Expr → Expr → Expr
```

Los constructores se introducen al definir el tipo.

También puedo definir **smart constructors**

```
val = Val  
add = Add
```

```
val x | x ≥ 0 = Val x
```

Tipos Algebraicos - Observadores

Dado el tipo:

```
data Expr where  
  Val  :: Int → Expr  
  Add  :: Expr → Expr → Expr
```

Puedo definir **observadores** (funciones) por casos, usando **pattern-matching** y **recursión**

```
eval :: Expr → Int  
eval (Val x)    = x  
eval (Add x y)  = eval x + eval y
```


Tipos Algebraicos - Observadores

Dado el tipo:

```
data Expr where
  Val  :: Int → Expr
  Add  :: Expr → Expr → Expr
```

Puedo definir **observadores** (funciones) por casos, usando **pattern-matching** y **recursión**

```
eval :: Expr → Int
eval (Val x)    = x
eval (Add x y)  = eval x + eval y
```

Los patrones satisfacen la gramática:

```
pat ::= _
      | variable
      | literal
      | (pat1, ..., patm)
      | pat : pat
      | C pat1 ... patn
      | var@pat
```

Observadores - Alto Orden

Múltiples observadores pueden compartir un **patrón** de recursión:

```
eval :: Expr → Int  
eval (Val x)    = x  
eval (Add x y) = eval x + eval y
```

```
cantOps :: Expr → Int  
cantOps (Val _)    = 1  
cantOps (Add x y) = cantOps x + cantOps y
```

```
ppExpr :: Expr → String  
ppExpr (Val x)    = show x  
ppExpr (Add x y) = ppExpr x ++ " + " ++ ppExpr y
```

Observadores - Alto Orden (2)

Puedo definir funciones de **alto orden** para capturar ese patrón

$$\text{foldExpr} :: (\text{Int} \rightarrow a) \rightarrow (a \rightarrow a \rightarrow a) \rightarrow \text{Expr} \rightarrow a$$
$$\text{foldExpr } fv _ (\text{Val } x) = fv \ x$$
$$\text{foldExpr } fv \ fa (\text{Add } x \ y) = fa \ (\text{foldExpr } fv \ fa \ x) \ (\text{foldExpr } fv \ fa \ y)$$

Observadores - Alto Orden (2)

Puedo definir funciones de **alto orden** para capturar ese patrón

$$\text{foldExpr} :: (\text{Int} \rightarrow a) \rightarrow (a \rightarrow a \rightarrow a) \rightarrow \text{Expr} \rightarrow a$$
$$\text{foldExpr } fv _ (\text{Val } x) = fv \ x$$
$$\text{foldExpr } fv \ fa (\text{Add } x \ y) = fa (\text{foldExpr } fv \ fa \ x) (\text{foldExpr } fv \ fa \ y)$$

Entonces

$$\text{eval} = \text{foldExpr } id \ (+)$$
$$\text{cantOps} = \text{foldExpr } (\text{const } 1) \ (+)$$
$$\text{ppExpr} = \text{foldExpr } show \ (\lambda ppx \ ppy \rightarrow ppx \ ++ \ " \ + \ " \ ++ \ ppy)$$

Alto Orden en Shallow Embedding

Agregamos variables a nuestro lenguaje de expresiones:

val :: *Int* → *Expr*

add :: *Expr* → *Expr* → *Expr*

var :: *String* → *Expr*

Alto Orden en Shallow Embedding

Agregamos variables a nuestro lenguaje de expresiones:

val :: *Int* → *Expr*

add :: *Expr* → *Expr* → *Expr*

var :: *String* → *Expr*

Nuestro evaluador debería poder aplicar el ambiente de variables:

eval :: *Expr* → [(*String*, *Int*)] → *Int*

Alto Orden en Shallow Embedding

Agregamos variables a nuestro lenguaje de expresiones:

$$\text{val} :: \text{Int} \rightarrow \text{Expr}$$
$$\text{add} :: \text{Expr} \rightarrow \text{Expr} \rightarrow \text{Expr}$$
$$\text{var} :: \text{String} \rightarrow \text{Expr}$$

Nuestro evaluador debería poder aplicar el ambiente de variables:

$$\text{eval} :: \text{Expr} \rightarrow [(\text{String}, \text{Int})] \rightarrow \text{Int}$$

Entonces el tipo *Expr* es:

$$\text{type Expr} = [(\text{String}, \text{Int})] \rightarrow \text{Int}$$

Alto Orden en Shallow Embedding

Agregamos variables a nuestro lenguaje de expresiones:

$$val :: Int \rightarrow Expr$$
$$add :: Expr \rightarrow Expr \rightarrow Expr$$
$$var :: String \rightarrow Expr$$

Nuestro evaluador debería poder aplicar el ambiente de variables:

$$eval :: Expr \rightarrow [(String, Int)] \rightarrow Int$$

Entonces el tipo *Expr* es:

$$\text{type } Expr = [(String, Int)] \rightarrow Int$$

y los constructores son funciones de alto orden

$$val\ x \quad = \lambda env \rightarrow x$$
$$add\ x\ y \quad = \lambda env \rightarrow x\ env + y\ env$$
$$var\ v \quad = \lambda env \rightarrow lookup\ v\ env$$

Type Classes

Podemos empaquetar la API del lenguaje en una type class:

```
class IExpr e where  
  val :: Int → e  
  add :: e → e → e  
  eval :: e → Int
```

Type Classes

Podemos empaquetar la API del lenguaje en una type class:

```
class IExpr e where  
  val :: Int → e  
  add :: e → e → e  
  eval :: e → Int
```

Definiendo instancias para cada implementación:

```
data Expr = Val Int | Add Expr Expr  
instance IExpr Expr where  
  val = Val  
  add = Add  
  eval = foldExpr id (+)
```

Type Classes

Podemos empaquetar la API del lenguaje en una type class:

```
class IExpr e where  
  val :: Int → e  
  add :: e → e → e  
  eval :: e → Int
```

Definiendo instancias para cada implementación:

```
data Expr = Val Int | Add Expr Expr  
  
instance IExpr Expr where  
  val = Val  
  add = Add  
  eval = foldExpr id (+)  
  
instance IExpr Int where  
  val n = n  
  add x y = x + y  
  eval e = e
```

Otro Ejemplo de EDSL - Regiones Geométricas

Consideremos un lenguaje que manipula **regiones geométricas** formado por las siguientes operaciones:

```
class Region r where  
  inRegion :: Point → r → Bool  
  circle    :: Radius → r  
  outside   :: r → r  
  union     :: r → r → r  
  intersect :: r → r → r
```

Otro Ejemplo de EDSL - Regiones Geométricas

Consideremos un lenguaje que manipula **regiones geométricas** formado por las siguientes operaciones:

```
class Region r where
  inRegion :: Point → r → Bool
  circle   :: Radius → r
  outside  :: r → r
  union    :: r → r → r
  intersect :: r → r → r
```

Ejemplo de un programa en el DSL:

```
aro :: Region r ⇒ Radius → Radius → r
aro r1 r2 = outside (circle r1) 'intersect' circle r2
```

Shallow embedding

Se captura directamente la semántica del dominio que manipula el DSL, en este caso *regiones*.

Una región geométrica se va a representar por la función característica del conjunto de puntos (dice que puntos están y cuales no).

```
data SRegion = R (Point → Bool)

instance Region SRegion where
  p 'inRegion' (R r)      = r p
  circle r                = R $ λp → magnitude p ≤ r
  outside (R r)           = R $ λp → ¬ (r p)
  (R r) 'union' (R r')    = R $ λp → r p ∨ r' p
  (R r) 'intersect' (R r') = R $ λp → r p ∧ r' p
```

Deep embedding

Se definen las formas de construir regiones a través de un tipo.

```
data DRegion = Circle    Radius
              | Outside  DRegion
              | Union     DRegion DRegion
              | Intersect DRegion DRegion
```

Deep embedding

Se definen las formas de construir regiones a través de un tipo.

```
data DRegion = Circle    Radius
              | Outside  DRegion
              | Union    DRegion DRegion
              | Intersect DRegion DRegion
```

y la instancia

```
instance Region DRegion where
  circle r          = Circle r
  outside r         = Outside r
  r 'union' r'      = Union r r'
  r 'intersect' r'  = Intersect r r'

  p 'inRegion' (Circle r)      = magnitude p ≤ r
  p 'inRegion' (Outside r)    = ¬ (p 'inRegion' r)
  p 'inRegion' (Union r r')   = p 'inRegion' r ∨ p 'inRegion' r'
  p 'inRegion' (Intersect r r') = p 'inRegion' r ∧ p 'inRegion' r'
```