# Abordaje Funcional a EDSLs

Alberto Pardo  Marcos Viera

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

ECI 2024

# Intérpretes Tagless-Final

# Embebiendo el Lenguaje

**Tagless-final** es una técnica de tipo shallow embedding para embeber lenguajes y sus interpretaciones.

# Embebiendo el Lenguaje

**Tagless-final** es una técnica de tipo shallow embedding para embeber lenguajes y sus interpretaciones.

Defino el lenguaje como una clase que contiene sus constructores:

```
class Expr e where
  val :: Int → e
  add :: e → e → e
```

# Embebiendo el Lenguaje

**Tagless-final** es una técnica de tipo shallow embedding para embeber lenguajes y sus interpretaciones.

Defino el lenguaje como una clase que contiene sus constructores:

```
class Expr e where
  val :: Int → e
  add :: e → e → e
```

```
expr1 :: Expr e ⇒ e
expr1 = add (val 8) (add (add (val 2) (val 1)) (val 4))
```

# Intérpretes

Defino las interpretaciones como instancias de la clase

# Intérpretes

Defino las interpretaciones como instancias de la clase

Evaluación:

```
data Eval = E Int

instance Expr Eval where
  val x             = E x
  add (E x) (E y) = E (x + y)
```

# Intérpretes

Defino las interpretaciones como instancias de la clase

Evaluación:

```
data Eval = E Int

instance Expr Eval where
  val x               = E x
  add (E x) (E y) = E (x + y)
```

Pretty-printing:

```
data PP = P String

instance Expr PP where
  val x               = P (show x)
  add (P x) (P y) = P ("(" ++ x ++ " + " ++ y ++ ")")
```

# Intérpretes

Defino las interpretaciones como instancias de la clase

Evaluación:

```
data Eval = E Int

instance Expr Eval where
  val x             = E x
  add (E x) (E y) = E (x + y)
```

Pretty-printing:

```
data PP = P String

instance Expr PP where
  val x             = P (show x)
  add (P x) (P y) = P ("(" ++ x ++ " + " ++ y ++ ")")
```

No necesito observadores

# Extensiones al Lenguaje

Puedo extender el lenguaje definiendo nuevas clases

```
class ExprMult e where
  mult :: e → e → e
```

# Extensiones al Lenguaje

Puedo extender el lenguaje definiendo nuevas clases

```
class ExprMult e where
  mult :: e → e → e
```

```
expr2 :: (Expr e, ExprMult e) ⇒ e
expr2 = add (mult expr1 (val 4)) (val 2)
```

# Extensiones al Lenguaje

Puedo extender el lenguaje definiendo nuevas clases

```
class ExprMult e where
  mult :: e → e → e
```

```
expr2 :: (Expr e, ExprMult e) ⇒ e
expr2 = add (mult expr1 (val 4)) (val 2)
```

Definiendo sus interpretaciones

```
instance ExprMult Eval where
  mult (E x) (E y) = E (x * y)
instance ExprMult PP where
  mult (P x) (P y) = P ("(" ++ x ++ " * " ++ y ++ ")")
```

# Lenguajes Tipados - Tagged

Si tenemos un lenguaje con distintos tipos

```
class Expr e where
  val    :: Int → e
  add    :: e → e → e
  isZero :: e → e
  ifE    :: e → e → e → e
```

# Lenguajes Tipados - Tagged

Si tenemos un lenguaje con distintos tipos

```
class Expr e where
  val    :: Int → e
  add    :: e → e → e
  isZero :: e → e
  ifE    :: e → e → e → e
```

¿Cómo resolvemos los problemas de tipado?

# Lenguajes Tipados - Tagged

Si tenemos un lenguaje con distintos tipos

```
class Expr e where
  val    :: Int → e
  add    :: e → e → e
  isZero :: e → e
  ifE    :: e → e → e → e
```

¿Cómo resolvemos los problemas de tipado?

Podríamos definir funciones parciales, o implementar el type-checking en el evaluador

```
data Res = RI Int | RB Bool   -- versión Tagged

instance Expr Res where
  val x                = RI x
  add (RI x) (RI y)    = RI (x + y)
  isZero (RI x)        = RB (x ≡ 0)
  ifE (RB c) (RI x) (RI y)  = RI $ if c then x else y
  ifE (RB c) (RB x) (RB y)  = RB $ if c then x else y
```

# Lenguajes Tipados - Tagless

O podemos codificar el sistema de tipos del lenguaje en la clase:

```
class TExpr e where
  valT    :: Int → e Int
  addT    :: e Int → e Int → e Int
  isZeroT :: e Int → e Bool
  ifT     :: e Bool → e t → e t → e t
```

# Lenguajes Tipados - Tagless

O podemos codificar el sistema de tipos del lenguaje en la clase:

```
class TExpr e where
  valT    :: Int → e Int
  addT    :: e Int → e Int → e Int
  isZeroT :: e Int → e Bool
  ifT     :: e Bool → e t → e t → e t
```

y usar el sistema de tipos del lenguaje anfitrión para chequearlo:

# Lenguajes Tipados - Tagless

O podemos codificar el sistema de tipos del lenguaje en la clase:

```
class TExpr e where
  valT    :: Int → e Int
  addT    :: e Int → e Int → e Int
  isZeroT :: e Int → e Bool
  ifT     :: e Bool → e t → e t → e t
```

y usar el sistema de tipos del lenguaje anfitrión para chequearlo:

```
exprT :: TExpr e ⇒ e Int
exprT = ifT (isZeroT (valT 2)) (valT 2) (valT 3)
```

# Lenguajes Tipados - Tagless

O podemos codificar el sistema de tipos del lenguaje en la clase:

```
class TExpr e where
  valT    :: Int → e Int
  addT    :: e Int → e Int → e Int
  isZeroT :: e Int → e Bool
  ifT     :: e Bool → e t → e t → e t
```

y usar el sistema de tipos del lenguaje anfitrión para chequearlo:

```
exprT :: TExpr e ⇒ e Int
exprT = ifT (isZeroT (valT 2)) (valT 2) (valT 3)
```

```
exprWrong = ifT (valT 1) (valT 2) (valT 3)   -- no compila
```

# Intérpretes Tipados

Ahora los intérpretes pueden usar la información del buen tipado

```
data TEval t = TE t

instance TExpr TEval where
  valT x                 = TE x
  addT (TE x) (TE y) = TE (x + y)
  isZeroT (TE x)         = TE (x ≡ 0)
  ifT (TE c) (TE x) (TE y) = TE (if c then x else y)
```

# Intérpretes Tipados

Ahora los intérpretes pueden usar la información del buen tipado

```
data TEval t = TE t

instance TExpr TEval where
  valT x                = TE x
  addT (TE x) (TE y) = TE (x + y)
  isZeroT (TE x)        = TE (x ≡ 0)
  ifT (TE c) (TE x) (TE y) = TE (if c then x else y)
```

o ignorarla a través de un phantom type

```
data TPP a = TP String

instance TExpr TPP where
  valT x                = TP (show x)
  addT (TP x) (TP y) = TP ("(" ++ x ++ " + " ++ y ++ ")")
  isZeroT (TP x)        = TP ("isZero(" ++ x ++ ")")
  ifT (TP c) (TP x) (TP y) = TP ("if " ++ c ++
                                 " then " ++ x ++
                                 " else " ++ y)
```