# Abordaje Funcional a EDSLs

Alberto Pardo     Marcos Viera

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

ECI 2024

# Embedded Domain Specific Languages

# Domain Specific Languages (1)

Un **lenguaje de dominio específico** (DSL) es un lenguaje de programación o especificación de expresividad limitada, especialmente diseñado para resolver problemas en un particular dominio.

Ejemplos:

- HTML
- VHDL (hardware)
- Mathematica, Maple
- SQL,XQuery (lenguajes de query)
- Yacc y Lex (para la generación de parsers)
- $\LaTeX$ (para producir documentos)
- DSLs para apl. financieras (`http://www.dslfin.org`)

# Domain Specific Languages (2)

Existen dos abordajes principales para implementar DSLs:

**Externo:** lenguaje standalone
Es necesario desarrollar:

- lexer, parser
- compilador
- herramientas

**Interno:** lenguaje implementado en el contexto de otro (*embebido*)

# DSLs embebidos

- Embedded DSLs (EDSLs) son DSLs implementados como bibliotecas específicas en lenguajes de propósito general que actuan como anfitrión (*host languages*)
- De esta manera el EDSL puede hacer uso de la infraestructura y facilidades existentes en el lenguaje anfitrión.
- La implementacion de un EDSL suele reducir el costo de desarrollo (se evita implementar lexer, parser, etc).
- Los lenguajes funcionales, en particular Haskell, son muy apropiados para la implementación de EDSLs.
- El manejo de errores suele ser un punto debil de los EDSLs.

# Ejemplos de EDSLs

Algunos ejemplos de EDSLs en Haskell:

- QuickCheck
- Sequence (finger trees)
- Streams
- HaXml (procesamiento de XML, HTML)
- Lava (hardware description)
- Parsec (parsing)
- Pretty printing
- Haskore (para componer música)

# Tipos de EDSLs

- **Shallow embedding**
  - Se captura directamente en un tipo de dato la semántica de los datos del dominio.
  - Dicha interpretación es fija.
  - Las operaciones del DSL manipulan directamente los valores del dominio.

# Tipos de EDSLs

- **Shallow embedding**
  - Se captura directamente en un tipo de dato la semántica de los datos del dominio.
  - Dicha interpretación es fija.
  - Las operaciones del DSL manipulan directamente los valores del dominio.

- **Deep embedding**
  - Las construcciones del DSL son representadas como términos de tipos de datos que corresponden a árboles de sintaxis abstracta (AST).
  - Estos términos son luego recorridos para su evaluación.
  - No hay una semántica fija, sino que se pueden definir diferentes interpretaciones.

# Ejemplo de EDSL

Consideremos un lenguaje que manipula expresiones aritméticas formado por las siguentes operaciones:

$$
\begin{array}{llll}
\mathit{val} & :: \mathit{Int} \rightarrow \mathit{Expr} & & \text{-- constructor} \\
\mathit{add} & :: \mathit{Expr} \rightarrow \mathit{Expr} \rightarrow \mathit{Expr} & & \text{-- constructor} \\
\mathit{eval} & :: \mathit{Expr} \rightarrow \mathit{Int} & & \text{-- observador}
\end{array}
$$

# Ejemplo de EDSL

Consideremos un lenguaje que manipula expresiones aritméticas formado por las siguentes operaciones:

$$
\begin{array}{llll}
\mathit{val} & :: \mathit{Int} \rightarrow \mathit{Expr} & & \text{-- constructor} \\
\mathit{add} & :: \mathit{Expr} \rightarrow \mathit{Expr} \rightarrow \mathit{Expr} & & \text{-- constructor} \\
\mathit{eval} & :: \mathit{Expr} \rightarrow \mathit{Int} & & \text{-- observador}
\end{array}
$$

Ejemplo de un programa en el DSL:

$$
\begin{array}{l}
\mathit{siete} :: \mathit{Expr} \\
\mathit{siete} = \mathit{add}\ (\mathit{val}\ 3)\ (\mathit{val}\ 4)
\end{array}
$$

$$
\begin{array}{l}
\mathit{doble} :: \mathit{Expr} \rightarrow \mathit{Expr} \\
\mathit{doble}\ e = \mathit{add}\ e\ e
\end{array}
$$

$$
\begin{array}{l}
\mathit{runDoble} :: \mathit{Expr} \rightarrow \mathit{Int} \\
\mathit{runDoble}\ e = \mathit{eval}\ (\mathit{doble}\ e)
\end{array}
$$

# Shallow embedding

Se captura directamente la semántica del dominio que manipula el DSL.

Para este tipo de expresiones aritméticas la representación por defecto es usar un entero, el cuál va a denotar el valor de la expresión.

```haskell
type Expr = Int
```

# Shallow embedding

Se captura directamente la semántica del dominio que manipula el DSL.

Para este tipo de expresiones aritméticas la representación por defecto es usar un entero, el cuál va a denotar el valor de la expresión.

$$\texttt{type}\ \mathit{Expr} = \mathit{Int}$$

**Constructores**

$$\begin{aligned} \mathit{val}\ n &= n \\ \mathit{add}\ e\ e' &= e + e' \end{aligned}$$

**Observador**

$$\mathit{eval}\ e = e$$

# Deep embedding (1)

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

$$
\begin{array}{l}
\mathtt{data}\ \mathit{Expr}\ \mathtt{where} \\
\quad \mathit{Val}\ \ :: \mathit{Int} \to \mathit{Expr} \\
\quad \mathit{Add} :: \mathit{Expr} \to \mathit{Expr} \to \mathit{Expr}
\end{array}
$$

# Deep embedding (1)

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

$$
\begin{array}{l}
\texttt{data}\ \mathit{Expr}\ \texttt{where} \\
\quad \mathit{Val}\ \ :: \mathit{Int} \to \mathit{Expr} \\
\quad \mathit{Add} :: \mathit{Expr} \to \mathit{Expr} \to \mathit{Expr}
\end{array}
$$

Operaciones de construcción (smart constructors):

$$
\begin{array}{l}
\mathit{val} :: \mathit{Int} \to \mathit{Expr} \\
\mathit{val}\ n = \mathit{Val}\ n \\
\\
\mathit{add} :: \mathit{Expr} \to \mathit{Expr} \to \mathit{Expr} \\
\mathit{add}\ e\ e' = \mathit{Add}\ e\ e'
\end{array}
$$

# Deep embedding (2)

El **observador** ahora hace las veces de función de interpretación.

$$
\begin{aligned}
&\mathit{eval} :: \mathit{Expr} \to \mathit{Int} \\
&\mathit{eval}\ (\mathit{Val}\ n) \qquad = n \\
&\mathit{eval}\ (\mathit{Add}\ e\ e') = \mathit{eval}\ e + \mathit{eval}\ e'
\end{aligned}
$$

# Que embedding elegir? (expression problem)

**Shallow embedding**

Pros: Es simple agregar nuevas construcciones al EDSL (por ejemplo, *mult*), mientras se puedan representar en el dominio de interpretación.

Cons: Agregar nuevas formas de interpretación (por ejemplo, hacer un pretty printing de las expresiones) puede implicar una reimplemantación completa.

# Que embedding elegir? (expression problem)

## Shallow embedding

Pros: Es simple agregar nuevas construcciones al EDSL (por ejemplo, *mult*), mientras se puedan representar en el dominio de interpretación.

Cons: Agregar nuevas formas de interpretación (por ejemplo, hacer un pretty printing de las expresiones) puede implicar una reimplemantación completa.

## Deep embedding

Pros: Es simple agregar un nuevo observador (por ejemplo, pretty printing).

Cons: Agregar nuevas construcciones al lenguaje (como *mult*) implica modificar el tipo del AST (el tipo *Expr*) y reimplementar todos los observadores (las funciones de interpretación).

# Razonamiento sobre el EDSL

A partir de la definición del EDSL en Haskell (tanto como shallow o deep embedding) es posible probar propiedades del EDSL.

Por ejemplo,

$$add\ e\ (add\ e'\ e'') \quad = \quad add\ (add\ e\ e')\ e''$$

$$add\ e\ e' \quad = \quad add\ e'\ e$$

# 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:

$$
\begin{array}{l}
\mathtt{data}\ \mathit{Expr}\ \mathtt{where} \\
\quad \mathit{Val}\ \ :: \mathit{Int} \to \mathit{Expr} \\
\quad \mathit{Add} :: \mathit{Expr} \to \mathit{Expr} \to \mathit{Expr}
\end{array}
$$

# Tipos Algebraicos

Notación de GADTs:

$$
\begin{array}{l}
\mathtt{data}\ \mathit{Expr}\ \mathtt{where} \\
\quad \mathit{Val}\ \ :: \mathit{Int} \to \mathit{Expr} \\
\quad \mathit{Add} :: \mathit{Expr} \to \mathit{Expr} \to \mathit{Expr}
\end{array}
$$

Notación clásica:

$$\mathtt{data}\ \mathit{Expr} = \mathit{Val}\ \mathit{Int} \mid \mathit{Add}\ \mathit{Expr}\ \mathit{Expr}$$

Notación de GADTs:

$$\begin{array}{l}\texttt{data}\ \mathit{Expr}\ \texttt{where}\\ \quad \mathit{Val}\ \ :: \mathit{Int} \rightarrow \mathit{Expr}\\ \quad \mathit{Add} :: \mathit{Expr} \rightarrow \mathit{Expr} \rightarrow \mathit{Expr}\end{array}$$

Notación clásica:

$$\texttt{data}\ \mathit{Expr} = \mathit{Val}\ \mathit{Int} \mid \mathit{Add}\ \mathit{Expr}\ \mathit{Expr}$$

En general:

$$\begin{array}{lll}\texttt{data}\ T\ a_1\ \ldots\ a_m & = & C_1\ t_{11}\ \ldots\ t_{1k_1}\\ & & \ldots\\ & \mid & C_n\ t_{n1}\ \ldots\ t_{nk_n}\end{array}$$

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

# Tipos Algebraicos (2)

Una manera simple de definir estructuras arborescentes:

$$\texttt{data}\ \mathit{Tree}\ a = \mathit{Leaf}\ a \mid \mathit{Fork}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)$$

# Tipos Algebraicos (2)

Una manera simple de definir estructuras arborescentes:

$$\mathtt{data}\ \mathit{Tree}\ a = \mathit{Leaf}\ a \mid \mathit{Fork}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)$$

$$\begin{array}{l} \mathtt{data}\ \mathit{Tree}\ a\ \mathtt{where} \\ \quad \mathit{Leaf} :: a \to \mathit{Tree}\ a \\ \quad \mathit{Fork} :: \mathit{Tree}\ a \to \mathit{Tree}\ a \to \mathit{Tree}\ a \end{array}$$

# Tipos Algebraicos (2)

Una manera simple de definir estructuras arborescentes:

$$\mathtt{data}\ \mathit{Tree}\ a = \mathit{Leaf}\ a \mid \mathit{Fork}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)$$

$$\begin{array}{l} \mathtt{data}\ \mathit{Tree}\ a\ \mathtt{where} \\ \quad \mathit{Leaf} :: a \rightarrow \mathit{Tree}\ a \\ \quad \mathit{Fork} :: \mathit{Tree}\ a \rightarrow \mathit{Tree}\ a \rightarrow \mathit{Tree}\ a \end{array}$$

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:

$$
\begin{array}{l}
\texttt{data}\ \mathit{Expr}\ \texttt{where} \\
\quad \mathit{Val}\ \ :: \mathit{Int} \to \mathit{Expr} \\
\quad \mathit{Add} :: \mathit{Expr} \to \mathit{Expr} \to \mathit{Expr}
\end{array}
$$

Los constructores se introducen al definir el tipo.

También puedo definir smart constructors

$$
\begin{array}{l}
\mathit{val}\ \ = \mathit{Val} \\
\mathit{add} = \mathit{Add}
\end{array}
$$

# 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
```

$$val\ x \mid x \geqslant 0 = Val\ x$$

# Tipos Algebraicos - Observadores

Dado el tipo:

$$\begin{array}{l}
\texttt{data}\ \mathit{Expr}\ \texttt{where} \\
\quad \mathit{Val}\ \ :: \mathit{Int} \to \mathit{Expr} \\
\quad \mathit{Add} :: \mathit{Expr} \to \mathit{Expr} \to \mathit{Expr}
\end{array}$$

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

$$\begin{array}{l}
\mathit{eval} :: \mathit{Expr} \to \mathit{Int} \\
\mathit{eval}\ (\mathit{Val}\ x) \qquad = x \\
\mathit{eval}\ (\mathit{Add}\ x\ y) = \mathit{eval}\ x + \mathit{eval}\ y
\end{array}$$

# Tipos Algebraicos - Observadores

Dado el tipo:

$$\mathtt{data}\ \mathit{Expr}\ \mathtt{where}$$
$$\quad \mathit{Val}\ :: \mathit{Int} \to \mathit{Expr}$$
$$\quad \mathit{Add} :: \mathit{Expr} \to \mathit{Expr} \to \mathit{Expr}$$

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

$$\begin{aligned}
&\mathit{eval} :: \mathit{Expr} \to \mathit{Int}\\
&\mathit{eval}\ (\mathit{Val}\ x) \quad\ = x\\
&\mathit{eval}\ (\mathit{Add}\ x\ y) = \mathit{eval}\ x + \mathit{eval}\ y
\end{aligned}$$

Los patrones satisfacen la gramática:

$$\begin{aligned}
\mathit{pat} ::= &\ _\\
\mid &\ \mathit{variable}\\
\mid &\ \mathit{literal}\\
\mid &\ (\mathit{pat}_1, ..., \mathit{pat}_m)\\
\mid &\ \mathit{pat} : \mathit{pat}\\
\mid &\ C\ \mathit{pat}_1\ ...\ \mathit{pat}_n\\
\mid &\ \mathit{var}@\mathit{pat}
\end{aligned}$$

# Observadores - Alto Orden

Múltiples observadores pueden compartir un patrón de recursión:

$$
\begin{aligned}
&\mathit{eval} :: \mathit{Expr} \to \mathit{Int}\\
&\mathit{eval}\ (\mathit{Val}\ x) \quad\ = x\\
&\mathit{eval}\ (\mathit{Add}\ x\ y) = \mathit{eval}\ x + \mathit{eval}\ y
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{cantOps} :: \mathit{Expr} \to \mathit{Int}\\
&\mathit{cantOps}\ (\mathit{Val}\ \_) \quad\ = 1\\
&\mathit{cantOps}\ (\mathit{Add}\ x\ y) = \mathit{cantOps}\ x + \mathit{cantOps}\ y
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{ppExpr} :: \mathit{Expr} \to \mathit{String}\\
&\mathit{ppExpr}\ (\mathit{Val}\ x) \quad\ = \mathit{show}\ x\\
&\mathit{ppExpr}\ (\mathit{Add}\ x\ y) = \mathit{ppExpr}\ x +\!\!+ \texttt{" + "} +\!\!+ \mathit{ppExpr}\ y
\end{aligned}
$$

# Observadores - Alto Orden (2)

Puedo definir funciones de alto orden para capturar ese patrón

$$
\begin{array}{l}
\mathit{foldExpr} :: (\mathit{Int} \to a) \to (a \to a \to a) \to \mathit{Expr} \to a \\
\mathit{foldExpr}\ \mathit{fv}\ \_\ (\mathit{Val}\ x) \quad\ = \mathit{fv}\ x \\
\mathit{foldExpr}\ \mathit{fv}\ \mathit{fa}\ (\mathit{Add}\ x\ y) = \mathit{fa}\ (\mathit{foldExpr}\ \mathit{fv}\ \mathit{fa}\ x)\ (\mathit{foldExpr}\ \mathit{fv}\ \mathit{fa}\ y)
\end{array}
$$

## Observadores - Alto Orden (2)

Puedo definir funciones de alto orden para capturar ese patrón

$$
\begin{aligned}
&foldExpr :: (Int \rightarrow a) \rightarrow (a \rightarrow a \rightarrow a) \rightarrow Expr \rightarrow a \\
&foldExpr\ fv\ \_\ (Val\ x) \quad = fv\ x \\
&foldExpr\ fv\ fa\ (Add\ x\ y) = fa\ (foldExpr\ fv\ fa\ x)\ (foldExpr\ fv\ fa\ y)
\end{aligned}
$$

Entonces

$$
\begin{aligned}
eval &= foldExpr\ id\ (+) \\
cantOps &= foldExpr\ (const\ 1)\ (+) \\
ppExpr &= foldExpr\ show\ (\lambda ppx\ ppy \rightarrow ppx +\!\!+ \texttt{" + "} +\!\!+ ppy)
\end{aligned}
$$

# Alto Orden en Shallow Embedding

Agregamos variables a nuestro lenguaje de expresiones:

$$
\begin{aligned}
&\mathit{val} :: \mathit{Int} \to \mathit{Expr} \\
&\mathit{add} :: \mathit{Expr} \to \mathit{Expr} \to \mathit{Expr} \\
&\mathit{var} :: \mathit{String} \to \mathit{Expr}
\end{aligned}
$$

# Alto Orden en Shallow Embedding

Agregamos variables a nuestro lenguaje de expresiones:

$$
\begin{aligned}
&val \;:: Int \rightarrow Expr \\
&add :: Expr \rightarrow Expr \rightarrow Expr \\
&var \;:: String \rightarrow Expr
\end{aligned}
$$

Nuestro evaluador debería poder aplicar el ambiente de variables:

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

# Alto Orden en Shallow Embedding

Agregamos variables a nuestro lenguaje de expresiones:

$$
\begin{aligned}
&\mathit{val} \;\; :: \mathit{Int} \to \mathit{Expr} \\
&\mathit{add} :: \mathit{Expr} \to \mathit{Expr} \to \mathit{Expr} \\
&\mathit{var} \;\; :: \mathit{String} \to \mathit{Expr}
\end{aligned}
$$

Nuestro evaluador debería poder aplicar el ambiente de variables:

$$\mathit{eval} :: \mathit{Expr} \to [(\mathit{String}, \mathit{Int})] \to \mathit{Int}$$

Entonces el tipo *Expr* es:

$$\texttt{type}\ \mathit{Expr} = [(\mathit{String}, \mathit{Int})] \to \mathit{Int}$$

# Alto Orden en Shallow Embedding

Agregamos variables a nuestro lenguaje de expresiones:

$$\begin{aligned} &val \;\; :: Int \rightarrow Expr \\ &add :: Expr \rightarrow Expr \rightarrow Expr \\ &var \;\; :: String \rightarrow Expr \end{aligned}$$

Nuestro evaluador debería poder aplicar el ambiente de variables:

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

Entonces el tipo *Expr* es:

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

y los constructores son funciones de alto orden

$$\begin{aligned} &val\ x &&= \lambda env \rightarrow x \\ &add\ x\ y &&= \lambda env \rightarrow x\ env + y\ env \\ &var\ v &&= \lambda env \rightarrow slookup\ v\ env \end{aligned}$$

# 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:

$$
\begin{array}{l}
\mathtt{class}\ \mathit{IExpr}\ e\ \mathtt{where} \\
\quad \mathit{val}\ \ :: \mathit{Int} \to e \\
\quad \mathit{add} :: e \to e \to e \\
\quad \mathit{eval} :: e \to \mathit{Int}
\end{array}
$$

Definiendo instancias para cada implementación:

$$
\mathtt{data}\ \mathit{Expr} = \mathit{Val}\ \mathit{Int} \mid \mathit{Add}\ \mathit{Expr}\ \mathit{Expr}
$$

$$
\begin{array}{l}
\mathtt{instance}\ \mathit{IExpr}\ \mathit{Expr}\ \mathtt{where} \\
\quad \mathit{val}\ \ = \mathit{Val} \\
\quad \mathit{add} = \mathit{Add} \\
\quad \mathit{eval} = \mathit{foldExpr}\ \mathit{id}\ (+)
\end{array}
$$

# Type Classes

Podemos empaquetar la API del lenguaje en una type class:

```haskell
class IExpr e where
  val  :: Int -> e
  add  :: e -> e -> e
  eval :: e -> Int
```

Definiendo instancias para cada implementación:

```haskell
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 siguentes 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 siguentes operaciones:

```haskell
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:

```haskell
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).

$$\mathtt{data}\ \mathit{SRegion} = R\ (\mathit{Point} \to \mathit{Bool})$$

$$\begin{array}{l} \mathtt{instance}\ \mathit{Region}\ \mathit{SRegion}\ \mathtt{where} \\ \quad \begin{array}{ll} p\ \text{`}\mathit{inRegion}\text{`}\ (R\ r) & = r\ p \\ \mathit{circle}\ r & = R\ \$\ \lambda p \to \mathit{magnitude}\ p \leqslant r \\ \mathit{outside}\ (R\ r) & = R\ \$\ \lambda p \to \neg\ (r\ p) \\ (R\ r)\ \text{`}\mathit{union}\text{`}\ (R\ r') & = R\ \$\ \lambda p \to r\ p \vee r'\ p \\ (R\ r)\ \text{`}\mathit{intersect}\text{`}\ (R\ r') & = R\ \$\ \lambda p \to r\ p \wedge r'\ p \end{array} \end{array}$$

# Deep embedding

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

$$
\begin{array}{lll}
\texttt{data}\ \mathit{DRegion} = & \mathit{Circle} & \mathit{Radius} \\
\mid & \mathit{Outside} & \mathit{DRegion} \\
\mid & \mathit{Union} & \mathit{DRegion}\ \mathit{DRegion} \\
\mid & \mathit{Intersect} & \mathit{DRegion}\ \mathit{DRegion}
\end{array}
$$

# Deep embedding

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

$$
\begin{array}{llll}
\mathtt{data}\ \mathit{DRegion} = & \mathit{Circle} & \mathit{Radius} \\
\mid & \mathit{Outside} & \mathit{DRegion} \\
\mid & \mathit{Union} & \mathit{DRegion}\ \mathit{DRegion} \\
\mid & \mathit{Intersect} & \mathit{DRegion}\ \mathit{DRegion}
\end{array}
$$

y la instancia

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathtt{instance}\ \mathit{Region}\ \mathit{DRegion}\ \mathtt{where}} \\
\quad \mathit{circle}\ r & = \mathit{Circle}\ r \\
\quad \mathit{outside}\ r & = \mathit{Outside}\ r \\
\quad r\ \text{`}\mathit{union}\text{`}\ r' & = \mathit{Union}\ r\ r' \\
\quad r\ \text{`}\mathit{intersect}\text{`}\ r' & = \mathit{Intersect}\ r\ r' \\
\quad p\ \text{`}\mathit{inRegion}\text{`}\ (\mathit{Circle}\ r) & = \mathit{magnitude}\ p \leqslant r \\
\quad p\ \text{`}\mathit{inRegion}\text{`}\ (\mathit{Outside}\ r) & = \neg\ (p\ \text{`}\mathit{inRegion}\text{`}\ r) \\
\quad p\ \text{`}\mathit{inRegion}\text{`}\ (\mathit{Union}\ r\ r') & = p\ \text{`}\mathit{inRegion}\text{`}\ r \vee p\ \text{`}\mathit{inRegion}\text{`}\ r' \\
\quad p\ \text{`}\mathit{inRegion}\text{`}\ (\mathit{Intersect}\ r\ r') & = p\ \text{`}\mathit{inRegion}\text{`}\ r \wedge p\ \text{`}\mathit{inRegion}\text{`}\ r'
\end{array}
$$

# Abordaje Funcional a EDSLs

Alberto Pardo     Marcos Viera

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

ECI 2024

# Functores Aplicativos

# Functor

Un **functor** puede entenderse como un constructor de tipo $f :: * \rightarrow *$ junto a una función de tipo

$$(a \rightarrow b) \rightarrow f\ a \rightarrow f\ b$$

que permite mapear/reemplazar los valores de tipo $a$ contenidos en una estructura de tipo $f\ a$ por valores de tipo $b$.

# Functor

En Haskell el concepto de functor es capturado por una clase:

$$\mathtt{class}\ \mathit{Functor}\ (f :: * \to *)\ \mathtt{where}$$
$$\quad \mathit{fmap} :: (a \to b) \to f\ a \to f\ b$$

# Functor

En Haskell el concepto de functor es capturado por una clase:

$$
\begin{aligned}
&\texttt{class}\ \mathit{Functor}\ (f :: * \to *)\ \texttt{where} \\
&\quad \mathit{fmap} :: (a \to b) \to f\ a \to f\ b
\end{aligned}
$$

Para ser efectivamente un functor la función *fmap* debe satisfacer las siguientes propiedades:

$$
\begin{aligned}
\mathit{fmap}\ \mathit{id} &= \mathit{id} \\
\mathit{fmap}\ (f \,.\, g) &= \mathit{fmap}\ f \,.\, \mathit{fmap}\ g
\end{aligned}
$$

que deberian ser chequeadas al definir cada instancia de la clase.

# Ejemplos

```
instance Functor [] where
  fmap = map
```

# Ejemplos

```haskell
instance Functor [] where
  fmap = map
```

```haskell
instance Functor Maybe where
  fmap f Nothing  = Nothing
  fmap f (Just a) = Just (f a)
```

# Ejemplos

```haskell
instance Functor [] where
  fmap = map
```

```haskell
instance Functor Maybe where
  fmap f Nothing  = Nothing
  fmap f (Just a) = Just (f a)
```

```haskell
instance Functor (Either a) where
  fmap f (Right x) = Right (f x)
  fmap f (Left x)  = Left x
```

# Ejemplos

$$\texttt{instance}\ \mathit{Functor}\ [\,]\ \texttt{where}$$
$$\quad \mathit{fmap} = \mathit{map}$$

$$\texttt{instance}\ \mathit{Functor}\ \mathit{Maybe}\ \texttt{where}$$
$$\quad \mathit{fmap}\ f\ \mathit{Nothing} = \mathit{Nothing}$$
$$\quad \mathit{fmap}\ f\ (\mathit{Just}\ a) = \mathit{Just}\ (f\ a)$$

$$\texttt{instance}\ \mathit{Functor}\ (\mathit{Either}\ a)\ \texttt{where}$$
$$\quad \mathit{fmap}\ f\ (\mathit{Right}\ x) = \mathit{Right}\ (f\ x)$$
$$\quad \mathit{fmap}\ f\ (\mathit{Left}\ x) \quad = \mathit{Left}\ x$$

$$\texttt{instance}\ \mathit{Functor}\ ((\to)\ r)\ \texttt{where}$$
$$\quad \mathit{fmap}\ f\ h = \lambda r \to f\ (h\ r) \quad \text{-- o sea, } f\,.\,h$$

# Modelando Error con `Maybe`

División segura

$$
\begin{aligned}
divM\ x\ y \mid y \not\equiv 0 \quad &= Just\ (x\ `div`\ y) \\
\mid otherwise &= Nothing
\end{aligned}
$$

# Modelando Error con `Maybe`

División segura

$$\begin{aligned} \mathit{divM}\ x\ y \mid y \not\equiv 0 &= \mathit{Just}\ (x\ \text{`}\mathit{div}\text{`}\ y) \\ \mid \mathit{otherwise} &= \mathit{Nothing} \end{aligned}$$

Con *fmap* puede aplicar una función pura al resultado de una división

$$\mathit{foo}\ x\ y = \mathit{fmap}\ (+2)\ (\mathit{divM}\ x\ y)$$

# Modelando Error con Maybe

División segura

$$\begin{aligned} divM\ x\ y \mid y \not\equiv 0 &= Just\ (x\ \text{`}div\text{`}\ y) \\ \mid otherwise &= Nothing \end{aligned}$$

Con *fmap* puede aplicar una función pura al resultado de una división

$$foo\ x\ y = fmap\ (+2)\ (divM\ x\ y)$$

en lugar de hacer:

$$\begin{aligned} foo\ x\ y = \mathtt{case}\ & divM\ x\ y\ \mathtt{of} \\ & Just\ r \rightarrow Just\ (r+2) \\ & Nothing \rightarrow Nothing \end{aligned}$$

# Functores Aplicativos

Los **functores aplicativos** son functores que permiten modelar efectos y aplicar funciones dentro del functor (lo que les da el mote de *aplicativos*).

```
class Functor f => Applicative f where
  pure  :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

# Functores Aplicativos

Los **functores aplicativos** son functores que permiten modelar efectos y aplicar funciones dentro del functor (lo que les da el mote de *aplicativos*).

$$
\begin{array}{l}
\texttt{class}\ \mathit{Functor}\ f \Rightarrow \mathit{Applicative}\ f\ \texttt{where} \\
\quad \mathit{pure}\ \ :: a \to f\ a \\
\quad (\texttt{<*>}) :: f\ (a \to b) \to f\ a \to f\ b
\end{array}
$$

Se debe cumplir que:

$$\mathit{fmap}\ f\ x = \mathit{pure}\ f\ \texttt{<*>}\ x$$

Sinónimo en *Applicative*:

$$
\begin{array}{l}
(\texttt{<\$>}) :: \mathit{Functor}\ f \Rightarrow (a \to b) \to f\ a \to f\ b \\
f\ \texttt{<\$>}\ t = \mathit{fmap}\ f\ t
\end{array}
$$

# Ejemplo: Maybe

```
instance Applicative Maybe where
  pure = Just
  (Just f) <*> (Just x) = Just (f x)
  _        <*> _        = Nothing
```

# Ejemplo: `Maybe`

```
instance Applicative Maybe where
  pure = Just
  (Just f) <*> (Just x) = Just (f x)
  _        <*> _        = Nothing
```

Puedo por ejemplo modelar expresiones con errores:

```
type Expr = Maybe Int

valE x     = pure x
addE x y = (+) <$> x <*> y
divE x y  = case (x, y) of
               (Just vx, Just vy) → divM vx vy
               _                  → Nothing
```

# Leyes de functores aplicativos

**Identidad**:

$$pure\ id \mathbin{<\!\!*\!\!>} u \equiv u$$

**Composición**:

$$pure\ (.) \mathbin{<\!\!*\!\!>} u \mathbin{<\!\!*\!\!>} v \mathbin{<\!\!*\!\!>} w \equiv u \mathbin{<\!\!*\!\!>} (v \mathbin{<\!\!*\!\!>} w)$$

**Homomorfismo**:

$$pure\ f \mathbin{<\!\!*\!\!>} pure\ x \equiv pure\ (f\ x)$$

**Intercambio**:

$$u \mathbin{<\!\!*\!\!>} pure\ x \equiv pure\ (\lambda f \rightarrow f\ x) \mathbin{<\!\!*\!\!>} u$$

## Leyes de functores aplicativos

**Identidad**:

$$pure\ id \mathbin{<\!\ast\!>} u \equiv u$$

**Composición**:

$$pure\ (.) \mathbin{<\!\ast\!>} u \mathbin{<\!\ast\!>} v \mathbin{<\!\ast\!>} w \equiv u \mathbin{<\!\ast\!>} (v \mathbin{<\!\ast\!>} w)$$

**Homomorfismo**:

$$pure\ f \mathbin{<\!\ast\!>} pure\ x \equiv pure\ (f\ x)$$

**Intercambio**:

$$u \mathbin{<\!\ast\!>} pure\ x \equiv pure\ (\lambda f \rightarrow f\ x) \mathbin{<\!\ast\!>} u$$

Si se cumplen, entonces se cumple:

$$fmap\ f\ x = pure\ f \mathbin{<\!\ast\!>} x$$

# Funciones sobre functores aplicativos

$$\begin{array}{l}
\mathit{sequenceA} :: \mathit{Applicative}\ f \Rightarrow [f\ a] \rightarrow f\ [a] \\
\mathit{sequenceA}\ [\,] \qquad\quad = \mathit{pure}\ [\,] \\
\mathit{sequenceA}\ (a : as) = (:)\ \texttt{<\$>}\ a\ \texttt{<*>}\ \mathit{sequenceA}\ as
\end{array}$$

$$\begin{array}{l}
\mathit{traverse} :: \mathit{Applicative}\ f \Rightarrow (a \rightarrow f\ b) \rightarrow [a] \rightarrow f\ [b] \\
\mathit{traverse}\ f = \mathit{sequenceA} . \mathit{fmap}\ f
\end{array}$$

que equivale a:

$$\begin{array}{l}
\mathit{traverse}\ f\ [\,] \qquad\quad = \mathit{pure}\ [\,] \\
\mathit{traverse}\ f\ (x : xs) = (:)\ \texttt{<\$>}\ f\ x\ \texttt{<*>}\ \mathit{traverse}\ f\ xs
\end{array}$$

# Alternative

En *Control.Applicative* también se define:

$$
\begin{array}{l}
\mathtt{class}\ \mathit{Applicative}\ f \Rightarrow \mathit{Alternative}\ f\ \mathtt{where} \\
\quad \mathit{empty} :: f\ a \\
\quad (<|>)\ :: f\ a \rightarrow f\ a \rightarrow f\ a \\
\quad \mathit{some}\ :: f\ a \rightarrow f\ [a] \quad \text{-- one or more} \\
\quad \mathit{many}\ :: f\ a \rightarrow f\ [a] \quad \text{-- zero or more}
\end{array}
$$

# Ejemplo de Alternative: Parsers

$$
\begin{array}{l}
\texttt{instance}\ \mathit{Applicative}\ (\mathit{Parser}\ s)\ \texttt{where} \\
\quad \mathit{pure} = \mathit{pSucceed} \\
\quad \text{<*>}\ = \text{<*>}
\end{array}
$$

$$
\begin{array}{l}
\texttt{instance}\ \mathit{Alternative}\ (\mathit{Parser}\ s)\ \texttt{where} \\
\quad \mathit{empty} \quad = \mathit{pFail} \\
\quad \text{<|>} \qquad = \text{<|>} \\
\quad \mathit{many} \quad = \mathit{pList} \\
\quad \mathit{some}\ p = (:)\ \text{<\$>}\ p\ \text{<*>}\ \mathit{pList}\ p
\end{array}
$$

donde

$$
\begin{array}{rl}
\mathit{pFail} & :: \mathit{Parser}\ s\ a \\
\mathit{pSuceed} & :: a \to \mathit{Parser}\ s\ a \\
\text{<*>} & :: \mathit{Parser}\ s\ (a \to b) \to \mathit{Parser}\ s\ a \to \mathit{Parser}\ s\ b \\
\text{<|>} & :: \mathit{Parser}\ s\ a \to \mathit{Parser}\ s\ a \to \mathit{Parser}\ s\ a \\
\mathit{pList} & :: \mathit{Parser}\ s\ a \to \mathit{Parser}\ s\ [a]
\end{array}
$$

```haskell
instance Applicative [] where
  pure x = [x]
  fs <*> xs = [f x | f <- fs, x <- xs]
```

Ejemplo:

$$[(+1),(+2)] \mathbin{<\!\!*\!\!>} [1,2,3]$$

# Ejemplo: listas

```
instance Applicative [] where
  pure x = [x]
  fs <*> xs = [f x | f ← fs, x ← xs]
```

Ejemplo:

$$[(+1),(+2)] \mathbin{<\!*\!>} [1,2,3]$$

*leadsto*

$$[2,3,4,3,4,5]$$

# Ejemplo: Either

```
data Either a b = Left a | Right b

instance Functor (Either e) where
  fmap f (Right a) = Right (f a)
  fmap f (Left e)  = Left e
```

# Ejemplo: Either

```haskell
data Either a b = Left a | Right b

instance Functor (Either e) where
  fmap f (Right a) = Right (f a)
  fmap f (Left e)  = Left   e
```

Una posible instancia de *Applicative*:

```haskell
instance Applicative (Either e) where
  pure = Right
  Right f <*> Right a = Right (f a)
  Right f <*> Left e  = Left e
  Left e  <*> _       = Left e
```

# Ejemplo: `Either`

```
data Either a b = Left a | Right b

instance Functor (Either e) where
  fmap f (Right a) = Right (f a)
  fmap f (Left e)  = Left   e
```

Una posible instancia de *Applicative*:

```
instance Applicative (Either e) where
  pure = Right
  Right f <*> Right a = Right (f a)
  Right f <*> Left e  = Left e
  Left e  <*> _       = Left e
```

Otra:

```
instance Monoid e => Applicative (Either e) where
  pure = Right
  Right f <*> Right a = Right (f a)
  Left e  <*> Right _ = Left e
  Right _ <*> Left e  = Left e
  Left e  <*> Left e' = Left (e `mappend` e')
```

# Composición

La clase de functores aplicativos es cerrada bajo la composición.

```
newtype (f :. g) a = Compose {getCompose :: f (g a)}

instance (Functor f, Functor g) => Functor (f :. g) where
  fmap f (Compose x) = Compose (fmap (fmap f) x)

instance (Applicative f, Applicative g)
            => Applicative (f :. g) where
  pure x = Compose (pure (pure x))
  Compose f <*> Compose x = Compose ((<*>) <$> f <*> x)
```

# Composición

La clase de functores aplicativos es cerrada bajo la composición.

```haskell
newtype (f :. g) a = Compose {getCompose :: f (g a)}

instance (Functor f, Functor g) => Functor (f :. g) where
  fmap f (Compose x) = Compose (fmap (fmap f) x)

instance (Applicative f, Applicative g)
             => Applicative (f :. g) where
  pure x = Compose (pure (pure x))
  Compose f <*> Compose x = Compose ((<*>) <$> f <*> x)
```

La composición de dos mónadas puede no ser una mónada, pero es un aplicativo.

# Abordaje Funcional a EDSLs

Alberto Pardo Marcos Viera

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

ECI 2024

# Parsers aplicativos

# Combinadores de parsing

Los combinadores de parsing forman un EDSL que es implementado usando un shallow embedding.

Están formados por dos grupos de funciones:

- Funciones básicas que sirven para reconocer determinados strings de entrada
- Un grupo de combinadores que permiten construir nuevos parsers a partir de otros.

# Parsers elementales

La mayoría de las bibliotecas de parsing están formadas por los siguientes 4 combinadores básicos:

| | |
|---|---|
| string vacío | *pSucceed* |
| terminales | *pSym s* |
| alternativa | *p* <\|> *q* |
| composición | *p* <*> *q* |

# El tipo de un parser

- Un parser puede ser entendido como una función que toma un string de entrada y retorna algo de tipo $a$:

$$\mathit{String} \rightarrow a$$

# El tipo de un parser

- Un parser puede ser entendido como una función que toma un string de entrada y retorna algo de tipo $a$:

$$String \to a$$

- Un parser podría ser ambiguo y retornar varios posibles valores, significando que puede haber varias formas de reconocer la entrada.

$$String \to [a]$$

# El tipo de un parser

- Un parser puede ser entendido como una función que toma un string de entrada y retorna algo de tipo $a$:

$$String \to a$$

- Un parser podría ser ambiguo y retornar varios posibles valores, significando que puede haber varias formas de reconocer la entrada.

$$String \to [a]$$

- Un parser podría no consumir toda la entrada y retornar además la parte de la entrada no consumida.

$$String \to [(a, String)]$$

# El tipo de un parser

En resumen,

$$\texttt{type}\ \mathit{Parser}\ a = \mathit{String} \to [(a, \mathit{String})]$$

# El tipo de un parser

En resumen,

$$\texttt{type}\ \mathit{Parser}\ a = \mathit{String} \rightarrow [(a, \mathit{String})]$$

Podemos abstraer el tipo *String*:

$$\texttt{type}\ \mathit{Parser}\ s\ a = \mathit{Eq}\ s \Rightarrow [s] \rightarrow [(a, [s])]$$

- En su lugar ponemos una lista de valores de tipo $s$.
- A los valores de tipo $s$ les vamos a requerir que sean comparables por igualdad (instancia de la clase *Eq*).

# Combinadores básicos

$$
\begin{array}{rll}
pFail & :: & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad Parser\ s\ a \\
pSucceed & :: & a \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \rightarrow Parser\ s\ a \\
pSym & :: & Eq\ s \Rightarrow s \qquad\qquad\qquad\qquad\qquad\qquad \rightarrow Parser\ s\ s \\
\texttt{<|>} & :: & Parser\ s\ a \qquad\qquad\ \ \rightarrow Parser\ s\ a \rightarrow Parser\ s\ a \\
\texttt{<*>} & :: & Parser\ s\ (a \rightarrow b) \rightarrow Parser\ s\ a \rightarrow Parser\ s\ b
\end{array}
$$

## Combinadores básicos

$$pFail :: Parser\ s\ a$$
$$pFail = \lambda cs \rightarrow [\,]$$

# Combinadores básicos

$$pFail :: Parser\ s\ a$$
$$pFail = \lambda cs \rightarrow [\,]$$

$$pSucceed \quad :: a \rightarrow Parser\ s\ a$$
$$pSucceed\ a = \lambda cs \rightarrow [(a, cs)]$$

# Combinadores básicos

$$\mathit{pFail} :: \mathit{Parser}\ s\ a$$
$$\mathit{pFail} = \lambda cs \rightarrow [\,]$$

$$\mathit{pSucceed} \quad :: a \rightarrow \mathit{Parser}\ s\ a$$
$$\mathit{pSucceed}\ a = \lambda cs \rightarrow [(a, cs)]$$

$$\begin{array}{l}
\mathit{pSym} \quad :: \mathit{Eq}\ s \Rightarrow s \rightarrow \mathit{Parser}\ s\ s \\
\mathit{pSym}\ s = \lambda cs \rightarrow \texttt{case}\ cs\ \texttt{of} \\
\qquad\qquad\qquad\qquad [\,] \qquad\quad \rightarrow [\,] \\
\qquad\qquad\qquad\qquad (c : cs') \rightarrow \texttt{if}\ c == s \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{then}\ [(c, cs')] \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{else}\ [\,]
\end{array}$$

# Combinadores básicos

$$(<|>) \quad :: \; \mathit{Parser}\ s\ a \rightarrow \mathit{Parser}\ s\ a \rightarrow \mathit{Parser}\ s\ a$$
$$p\ <|>\ q = \lambda cs \rightarrow p\ cs +\!\!+\ q\ cs$$

# Combinadores básicos

$$(\texttt{<|>}) \quad :: \; Parser\ s\ a \rightarrow Parser\ s\ a \rightarrow Parser\ s\ a$$
$$p\ \texttt{<|>}\ q = \lambda cs \rightarrow p\ cs +\!\!\!+\ q\ cs$$

$$(\texttt{<*>}) \quad :: \; Parser\ s\ (a \rightarrow b) \rightarrow Parser\ s\ a \rightarrow Parser\ s\ b$$
$$\begin{aligned}(p \texttt{<*>} q)\ cs = [(f\ a, cs'') \mid\ & (f, cs') \leftarrow p\ cs \\ , & (a, cs'') \leftarrow q\ cs']\end{aligned}$$

# Ejemplos de parsers (1)

- Reconocer una 'A' y retornar una 'B':

# Ejemplos de parsers (1)

- Reconocer una 'A' y retornar una 'B':

$$pA2B = pSucceed\ (\lambda\_ \rightarrow \texttt{'B'})\ \texttt{<*>}\ pSym\ \texttt{'A'}$$

# Ejemplos de parsers (1)

- Reconocer una 'A' y retornar una 'B':

$$pA2B = pSucceed\ (\lambda\_ \rightarrow \texttt{'B'})\ \texttt{<*>}\ pSym\ \texttt{'A'}$$

- Reconocer una 'A', seguida de una 'B', y retornar ambos caracteres en un par.

# Ejemplos de parsers (1)

- Reconocer una 'A' y retornar una 'B':

$$pA2B = pSucceed\ (\lambda\_ \rightarrow \texttt{'B'})\ \texttt{<*>}\ pSym\ \texttt{'A'}$$

- Reconocer una 'A', seguida de una 'B', y retornar ambos caracteres en un par.

$$pAB = pSucceed\ (,)\ \texttt{<*>}\ pSym\ \texttt{'A'}\ \texttt{<*>}\ pSym\ \texttt{'B'}$$

# Ejemplos de parsers (2)

- Parser que retorna una lista de valores de tipo $a$. Toma como parámetro un parser que retorna un $a$.

# Ejemplos de parsers (2)

- Parser que retorna una lista de valores de tipo $a$. Toma como parámetro un parser que retorna un $a$.

$$
\begin{aligned}
&pList :: Parser\ s\ a \rightarrow Parser\ s\ [a] \\
&pList\ p = pSucceed\ (:) \mathbin{<\!*\!>} p \mathbin{<\!*\!>} pList\ p \\
&\qquad\qquad \mathbin{<\!|\!>} \\
&\qquad\qquad pSucceed\ [\,]
\end{aligned}
$$

# Ejemplos de parsers (2)

- Parser que retorna una lista de valores de tipo $a$. Toma como parámetro un parser que retorna un $a$.

$$
\begin{aligned}
&pList :: Parser\ s\ a \rightarrow Parser\ s\ [a] \\
&pList\ p = pSucceed\ (:)\ \texttt{<*>}\ p\ \texttt{<*>}\ pList\ p \\
&\qquad\qquad \texttt{<|>} \\
&\qquad\qquad pSucceed\ [\,]
\end{aligned}
$$

- Parser que reconoce un string de la forma $(AB)^*$.

# Ejemplos de parsers (2)

- Parser que retorna una lista de valores de tipo $a$. Toma como parámetro un parser que retorna un $a$.

$$
\begin{aligned}
&pList :: Parser\ s\ a \rightarrow Parser\ s\ [a] \\
&pList\ p = pSucceed\ (:) <\!*\!> p <\!*\!> pList\ p \\
&\qquad\qquad <|> \\
&\qquad\qquad pSucceed\ [\,]
\end{aligned}
$$

- Parser que reconoce un string de la forma $(AB)^*$.

$$pListAB = pList\ pAB$$

# Otros combinadores útiles

$$(\texttt{<\$>}) \quad :: (a \to b) \to \mathit{Parser}\ s\ a \to \mathit{Parser}\ s\ b$$
$$f\ \texttt{<\$>}\ p = \mathit{pSucceed}\ f\ \texttt{<*>}\ p$$

$$\mathit{opt} \quad :: \mathit{Parser}\ s\ a \to a \to \mathit{Parser}\ s\ a$$
$$p\ \text{`}\mathit{opt}\text{`}\ a = p\ \texttt{<|>}\ \mathit{pSucceed}\ a$$

$$\mathit{pSat} \quad :: (s \to \mathit{Bool}) \to \mathit{Parser}\ s\ s$$
$$\begin{array}{l} \mathit{pSat}\ p = \lambda cs \to \texttt{case}\ cs\ \texttt{of} \\ \qquad\qquad [\,] \qquad \to [\,] \\ \qquad\qquad (c : cs') \to \texttt{if}\ p\ c \\ \qquad\qquad\qquad\qquad \texttt{then}\ [(c, cs')] \\ \qquad\qquad\qquad\qquad \texttt{else}\ [\,] \end{array}$$

# Ejemplos

- Definición de *pAB* usando `<$>`:

$$pAB = (,) \texttt{<\$>}\ pSym\ \texttt{'A'}\ \texttt{<*>}\ pSym\ \texttt{'B'}$$

# Ejemplos

- Definición de *pAB* usando `<$>`:

$$pAB = (,)\ \texttt{<\$>}\ pSym\ \texttt{'A'}\ \texttt{<*>}\ pSym\ \texttt{'B'}$$

- Definición de *pSym* usando *pSat*:

$$pSym\ a = pSat\ (== a)$$

# Ejemplos

- Definición de *pAB* usando <$>:

$$pAB = (,) \mathtt{<\$>}\ pSym\ \mathtt{'A'}\ \mathtt{<*>}\ pSym\ \mathtt{'B'}$$

- Definición de *pSym* usando *pSat*:

$$pSym\ a = pSat\ (== a)$$

- Reconocer un dígito:

$$\begin{array}{l} pDigit = pSat\ isDigit \\ \quad \mathtt{where} \\ \qquad isDigit\ c = (c \geqslant \mathtt{'0'}) \wedge (c \leqslant \mathtt{'9'}) \end{array}$$

# Ejemplos

- Definición de *pAB* usando <$>:

$$pAB = (,) \texttt{<\$>}\ pSym\ \texttt{'A'}\ \texttt{<*>}\ pSym\ \texttt{'B'}$$

- Definición de *pSym* usando *pSat*:

$$pSym\ a = pSat\ (== a)$$

- Reconocer un dígito:

$$\begin{array}{l} pDigit = pSat\ isDigit \\ \quad \textbf{where} \\ \qquad isDigit\ c = (c \geqslant \texttt{'0'}) \wedge (c \leqslant \texttt{'9'}) \end{array}$$

- Definición de *pList* usando <$> y *opt*:

$$pList\ p = (:)\ \texttt{<\$>}\ p\ \texttt{<*>}\ pList\ p\ \text{`}opt\text{`}\ [\,]$$

# Selección de resultados de parsers

$$(<\!*) \quad :: \textit{Parser}\ s\ a \rightarrow \textit{Parser}\ s\ b \rightarrow \textit{Parser}\ s\ a$$
$$p <\!*\ q = (\lambda x\ \_ \rightarrow x) <\!\$\!> p <\!*\!> q$$

$$(*\!>) \quad :: \textit{Parser}\ s\ a \rightarrow \textit{Parser}\ s\ b \rightarrow \textit{Parser}\ s\ b$$
$$p *\!> q = (\lambda\_\ y \rightarrow y) <\!\$\!> p <\!*\!> q$$

$$(<\!\$) \quad :: a \rightarrow \textit{Parser}\ s\ b \rightarrow \textit{Parser}\ s\ a$$
$$a <\!\$\ q = \textit{pSucceed}\ a <\!*\ q$$

# Selección de resultados de parsers

$$(<\!*) \quad :: \; Parser\ s\ a \to Parser\ s\ b \to Parser\ s\ a$$
$$p <\!* q = (\lambda x\ \_ \to x) <\!\$\!> p <\!*\!> q$$

$$(*\!>) \quad :: \; Parser\ s\ a \to Parser\ s\ b \to Parser\ s\ b$$
$$p *\!> q = (\lambda\_\ y \to y) <\!\$\!> p <\!*\!> q$$

$$(<\!\$) \quad :: \; a \to Parser\ s\ b \to Parser\ s\ a$$
$$a <\!\$ q = pSucceed\ a <\!* q$$

**Ejemplo.** Reconocer algo entre paréntesis.

$$pParens\ p = pSym\ \text{'('}\ *\!> p <\!* pSym\ \text{')'}$$

# Abordaje Funcional a EDSLs

Alberto Pardo Marcos Viera

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

ECI 2024

# Parsers monádicos

# Mónada de Parsing

$$\texttt{newtype}\ Parser\ a = P\ (String \to [(a, String)])$$

$$runP :: Parser\ a \to String \to [(a, String)]$$
$$runP\ (P\ p) = p$$

$$\begin{aligned}
&\texttt{instance}\ Monad\ Parser\ \texttt{where}\\
&\quad return\ a \quad\ \ = P\ \$\ \lambda cs \to [(a, cs)]\\
&\quad (P\ p) \gg\!= f = P\ \$\ \lambda cs \to\\
&\qquad\qquad\qquad concat\ [runP\ (f\ a)\ cs' \mid (a, cs') \leftarrow p\ cs]
\end{aligned}$$

# Parsing: combinadores básicos

$$
\begin{array}{l}
pFail :: Parser\ a \\
pFail = P\ \$\ \lambda cs \rightarrow [\,]
\end{array}
$$

$$
\begin{array}{l}
item :: Parser\ Char \\
item = P\ \$\ \lambda cs \rightarrow \mathtt{case}\ cs\ \mathtt{of} \\
\qquad\qquad\qquad\qquad\qquad \texttt{""} \qquad\ \rightarrow [\,] \\
\qquad\qquad\qquad\qquad\qquad (c : cs) \rightarrow [(c, cs)]
\end{array}
$$

$$
\begin{array}{l}
pSat :: (Char \rightarrow Bool) \rightarrow Parser\ Char \\
pSat\ p = \mathtt{do}\ c \leftarrow item \\
\qquad\qquad\quad \mathtt{if}\ p\ c\ \mathtt{then}\ return\ c\ \mathtt{else}\ pFail
\end{array}
$$

$$
\begin{array}{l}
pSym :: Char \rightarrow Parser\ Char \\
pSym\ c = pSat\ (== c)
\end{array}
$$

# Parsing: Alternativa

$$(\texttt{<|>}) :: \mathit{Parser}\ a \to \mathit{Parser}\ a \to \mathit{Parser}\ a$$
$$(P\ p)\ \texttt{<|>}\ (P\ q) = P\ \$\ \lambda cs \to p\ cs +\!\!+\ q\ cs$$

# Parsing: Alternativa

$$(<|>) :: Parser\ a \rightarrow Parser\ a \rightarrow Parser\ a$$
$$(P\ p)\ <|>\ (P\ q) = P\ \$\ \lambda cs \rightarrow p\ cs +\!\!+\ q\ cs$$

Otra forma de definir el operador de alternativa:

$$\begin{array}{l} (P\ p)\ <|>\ (P\ q) = P\ \$\ \lambda cs \rightarrow \mathtt{case}\ p\ cs +\!\!+\ q\ cs\ \mathtt{of} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \begin{array}{ll} [\,] & \rightarrow [\,] \\ (x : xs) & \rightarrow [x] \end{array} \end{array}$$

# many y some

$p^*$ (many) cero o más veces p

$$
\begin{aligned}
&pList :: Parser\ a \rightarrow Parser\ [a] \\
&pList\ p = \mathtt{do}\ a \leftarrow p \\
&\qquad\qquad\qquad as \leftarrow pList\ p \\
&\qquad\qquad\qquad return\ (a : as) \\
&\qquad\qquad\quad <|> \\
&\qquad\qquad\quad return\ [\,]
\end{aligned}
$$

$p^+$ (some) una o más veces p

$$
\begin{aligned}
&pListP :: Parser\ a \rightarrow Parser\ [a] \\
&pListP\ p = \mathtt{do}\ a \leftarrow p \\
&\qquad\qquad\qquad as \leftarrow pList\ p \\
&\qquad\qquad\qquad return\ (a : as)
\end{aligned}
$$

# Ejemplo: digits

$$
\begin{aligned}
&digit :: Parser\ Int \\
&digit = \mathtt{do}\ c \leftarrow pSat\ isDigit \\
&\qquad\qquad\quad return\ (ord\ c - ord\ \text{'0'})
\end{aligned}
$$

$$
isDigit\ c = (c \geqslant \text{'0'}) \wedge (c \leqslant \text{'9'})
$$

$$
\begin{aligned}
&digits :: Parser\ [Int] \\
&digits = pListP\ digit
\end{aligned}
$$

$$
\begin{aligned}
&sumDigits :: Parser\ Int \\
&sumDigits = \mathtt{do}\ ds \leftarrow digits \\
&\qquad\qquad\qquad\quad return\ (sum\ ds)
\end{aligned}
$$

# Ejemplo: `number`

$$
\begin{aligned}
&\mathit{number} :: \mathit{Parser\ Int} \\
&\mathit{number} = \mathtt{do}\ d \leftarrow \mathit{digit} \\
&\qquad\qquad\quad\ \mathit{number}'\ d
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{number}' :: \mathit{Int} \rightarrow \mathit{Parser\ Int} \\
&\mathit{number}'\ n = \ \mathtt{do}\ d \leftarrow \mathit{digit} \\
&\qquad\qquad\qquad\quad\ \mathit{number}'\ (n * 10 + d) \\
&\qquad\qquad\quad <\mid> \\
&\qquad\qquad\quad \mathit{return}\ n
\end{aligned}
$$

# Ejemplo: number

$$
\begin{array}{l}
number :: Parser\ Int \\
number = \mathbf{do}\ d \leftarrow digit \\
\qquad\qquad\qquad number'\ d
\end{array}
$$

$$
\begin{array}{l}
number' :: Int \rightarrow Parser\ Int \\
number'\ n = \ \mathbf{do}\ d \leftarrow digit \\
\qquad\qquad\qquad\quad number'\ (n * 10 + d) \\
\qquad\qquad\quad <|> \\
\qquad\qquad\quad return\ n
\end{array}
$$

Esto equivale a la siguente definición:

$$
\begin{array}{l}
number = \mathbf{do}\ (d : ds) \leftarrow digits \\
\qquad\qquad\qquad return\ (foldl\ (\oplus)\ d\ ds) \\
\qquad\qquad \mathbf{where} \\
\qquad\qquad\quad n \oplus d = n * 10 + d
\end{array}
$$

# Parser para expresiones

Queremos parsear una expresión y retornar el correspondiente árbol de sintaxis abstracta (AST) de tipo:

$$\texttt{data}\ \mathit{Expr} = \mathit{Val}\ \mathit{Int} \mid \mathit{Add}\ \mathit{Expr}\ \mathit{Expr}$$

# Parser para expresiones

Queremos parsear una expresión y retornar el correspondiente árbol de sintaxis abstracta (AST) de tipo:

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

Que tal este parser?

```
expr :: Parser Expr
expr = do e1 ← expr
          pSym '+'
          e2 ← expr
          return (Add e1 e2)
       <|>
       do n ← number
          return (Val n)
```

# Parser para expresiones

Queremos parsear una expresión y retornar el correspondiente árbol de sintaxis abstracta (AST) de tipo:

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

Que tal este parser?

```
expr :: Parser Expr
expr = do e1 ← expr
          pSym '+'
          e2 ← expr
          return (Add e1 e2)
       <|>
       do n ← number
          return (Val n)
```

**Diverge!** La recursividad a la izquierda hace que entre en loop

# Parser para expresiones

Para eliminar la recursión a la izquierda debemos basarnos en la siguiente gramática:

$$e ::= n + e \mid n$$

# Parser para expresiones

Para eliminar la recursión a la izquierda debemos basarnos en la siguiente gramática:

$$e ::= n + e \mid n$$

El parser queda entonces de la siguiente forma:

```
expr :: Parser Expr
expr = do n ← number
          pSym '+'
          e ← expr
          return (Add (Val n) e)
       <|>
       do n ← number
          return (Val n)
```

## Parsing y evaluación de expresiones

$$\begin{aligned} \mathit{evalExpr} = \mathtt{do}\ & e \leftarrow \mathit{expr} \\ & \mathit{return}\ (\mathit{eval}\ e) \end{aligned}$$

$$\mathit{evalExpr} = \mathbf{do}\ e \leftarrow \mathit{expr}$$
$$\qquad\qquad \mathit{return}\ (\mathit{eval}\ e)$$

Es posible fusionar las definiciones de *eval* y *expr* y obtener una definición de *evalExp* que computa directamente el valor de la expresión parseada sin generar el AST intermedio:

```
evalExpr :: Parser Int
evalExpr = do n ← number
              pSym '+'
              m ← evalExpr
              return (n + m)
           <|>
           number
```

# Parser de un nano XML

```
data XML = Tag Char [XML]

xml :: Parser XML
xml = do    -- se parsea el tag de apertura
        pSym '<'
        name ← item
        pSym '>'
          -- se parsea la lista de XMLs internos
        xmls ← pList xml
          -- se parsea el tag de cierre
        pSym '<'
        pSym '/'
        pSym name   -- se utiliza nombre del tag de apertura
        pSym '>'
        return (Tag name xmls)
```

# Abordaje Funcional a EDSLs

Alberto Pardo     Marcos Viera

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

ECI 2024

## Modelado de efectos computacionales por mónadas

# Expresiones con Fallas

$$\texttt{data}\ \mathit{Expr} = \mathit{Val}\ \mathit{Int} \mid \mathit{Add}\ \mathit{Expr}\ \mathit{Expr} \mid \mathit{Div}\ \mathit{Expr}\ \mathit{Expr}$$

La operación de división debe controlar el caso excepcional de *división por cero*.

$$\texttt{data}\ \mathit{Maybe}\ a = \mathit{Just}\ a \mid \mathit{Nothing}$$

$$\begin{array}{ll}
\mathit{divM} & :: \mathit{Int} \to \mathit{Int} \to \mathit{Maybe}\ \mathit{Int} \\
a\ \text{`}\mathit{divM}\text{`}\ b & = \texttt{if}\ b \equiv 0\ \texttt{then}\ \mathit{Nothing} \\
& \phantom{= \texttt{if}\ b \equiv 0\ } \texttt{else}\ \mathit{Just}\ (a\ \text{`}\mathit{div}\text{`}\ b)
\end{array}$$

# Evaluador con Fallas

$$
\begin{array}{lll}
eval & :: Expr \to Maybe\ Int & \\
eval\ (Val\ n) & = Just\ n & \\
eval\ (Add\ x\ y) & = \mathbf{case}\ eval\ x\ \mathbf{of} & \\
 & \quad Nothing \to Nothing & \\
 & \quad Just\ a \quad \to \mathbf{case}\ eval\ y\ \mathbf{of} & \\
 & \qquad\qquad Nothing \to Nothing & \\
 & \qquad\qquad Just\ b \quad \to Just\ (a + b) & \\
eval\ (Div\ x\ y) & = \mathbf{case}\ eval\ x\ \mathbf{of} & \\
 & \quad Nothing \to Nothing & \\
 & \quad Just\ a \quad \to \mathbf{case}\ eval\ y\ \mathbf{of} & \\
 & \qquad\qquad Nothing \to Nothing & \\
 & \qquad\qquad Just\ b \quad \to a\ `divM`\ b &
\end{array}
$$

# Evaluador con Fallas - Applicative

```
eval             :: Expr → Maybe Int
eval (Val n)     = pure n
eval (Add x y) = (+) <$> eval x <*> eval y
eval (Div x y)  = case eval x of
                       Nothing → Nothing
                       Just a    → case eval y of
                                          Nothing → Nothing
                                          Just b    → a ‘divM‘ b
```

# Evaluador con Fallas - Applicative

```haskell
eval              :: Expr → Maybe Int
eval (Val n)      = pure n
eval (Add x y)    = (+) <$> eval x <*> eval y
eval (Div x y)    = case eval x of
                      Nothing → Nothing
                      Just a  → case eval y of
                                  Nothing → Nothing
                                  Just b  → a `divM` b
```

No puedo representar la división con Functores Aplicativos, necesito el resultado de una computación para determinar el siguiente efecto.

# Capturemos patrones

Definamos:

$$
\begin{array}{l}
\mathit{return} \quad :: a \rightarrow \mathit{Maybe}\ a \\
\mathit{return}\ a = \mathit{Just}\ a
\end{array}
$$

$$
\begin{array}{l}
(\gg=) \quad :: \mathit{Maybe}\ a \rightarrow (a \rightarrow \mathit{Maybe}\ b) \rightarrow \mathit{Maybe}\ b \\
m \gg= f = \mathtt{case}\ m\ \mathtt{of} \\
\qquad\qquad \mathit{Nothing} \rightarrow \mathit{Nothing} \\
\qquad\qquad \mathit{Just}\ a \quad \rightarrow f\ a
\end{array}
$$

Entonces,

$$
\begin{array}{ll}
\mathit{eval} & :: \mathit{Expr} \rightarrow \mathit{Maybe\ Int} \\
\mathit{eval}\ (\mathit{Val}\ n) & = \mathit{return}\ n \\
\mathit{eval}\ (\mathit{Add}\ x\ y) & = \mathit{eval}\ x \gg\!= (\lambda a \rightarrow \mathit{eval}\ y \gg\!= (\lambda b \rightarrow \mathit{return}\ (a + b))) \\
\mathit{eval}\ (\mathit{Div}\ x\ y) & = \mathit{eval}\ x \gg\!= (\lambda a \rightarrow \mathit{eval}\ y \gg\!= (\lambda b \rightarrow a\ `\mathit{divM}`\ b))
\end{array}
$$

# Evaluador con Fallas

$$
\begin{aligned}
&eval && :: Expr \to Maybe\ Int \\
&eval\ (Val\ n) && = return\ n \\
&eval\ (Add\ x\ y) && = eval\ x \ggg\!= \lambda a \to \\
& && \quad\ eval\ y \ggg\!= \lambda b \to \\
& && \quad\ return\ (a + b) \\
&eval\ (Div\ x\ y) && = eval\ x \ggg\!= \lambda a \to \\
& && \quad\ eval\ y \ggg\!= \lambda b \to \\
& && \quad\ a\ `divM`\ b
\end{aligned}
$$

# La clase Monad

$$
\begin{array}{l}
\texttt{class}\ \mathit{Applicative}\ m \Rightarrow \mathit{Monad}\ m\ \texttt{where} \\
\quad (\gg\!=) \quad :: m\ a \to (a \to m\ b) \to m\ b \\
\quad (\gg) \quad\ \ :: m\ a \to m\ b \to m\ b \\
\quad \mathit{return}\ :: a \to m\ a
\end{array}
$$

$$
m \gg k = m \gg\!= \lambda\_ \to k
$$

# La clase Monad

$$
\begin{array}{l}
\texttt{class}\ \mathit{Applicative}\ m \Rightarrow \mathit{Monad}\ m\ \texttt{where} \\
\quad (\gg\!=) \quad :: m\ a \to (a \to m\ b) \to m\ b \\
\quad (\gg) \quad\ \ :: m\ a \to m\ b \to m\ b \\
\quad \mathit{return}\ :: a \to m\ a
\end{array}
$$

$$m \gg k = m \gg\!= \lambda\_ \to k$$

Toda mónada es un functor aplicativo que cumple:

- $\mathit{pure} = \mathit{return}$
- $m1\ \texttt{<*>}\ m2 = m1 \gg\!= (\lambda f \to m2 \gg\!= (\lambda x \to \mathit{return}\ (f\ x)))$

# La clase Monad

$$
\begin{array}{l}
\texttt{class}\ \mathit{Applicative}\ m \Rightarrow \mathit{Monad}\ m\ \texttt{where} \\
\quad (\gg\!=)\quad :: m\ a \to (a \to m\ b) \to m\ b \\
\quad (\gg)\quad :: m\ a \to m\ b \to m\ b \\
\quad \mathit{return}\ :: a \to m\ a
\end{array}
$$

$$m \gg k = m \gg\!= \lambda\_ \to k$$

Toda mónada es un functor aplicativo que cumple:

- $\mathit{pure} = \mathit{return}$
- $m1 \ \texttt{<*>}\ m2 = m1 \gg\!= (\lambda f \to m2 \gg\!= (\lambda x \to \mathit{return}\ (f\ x)))$

No todo functor aplicativo es una mónada

# Mónada Maybe

```
data Maybe a = Just a | Nothing

instance Monad Maybe where
  return  = Just
  m >>= k = case m of
              Just x  -> k x
              Nothing -> Nothing
```

# Mónada Maybe

```
data Maybe a = Just a | Nothing

instance Monad Maybe where
  return  = Just
  m >>= k = case m of
              Just x  -> k x
              Nothing -> Nothing

instance Functor Maybe where
  fmap f Nothing  = Nothing
  fmap f (Just a) = Just (f a)

instance Applicative Maybe where
  pure = Just
  (Just f) <*> (Just x) = Just (f x)
  _        <*> _        = Nothing
```

# Mónada Maybe

$$\texttt{data}\ \mathit{Maybe}\ a = \mathit{Just}\ a \mid \mathit{Nothing}$$

$$\begin{array}{l}
\texttt{instance}\ \mathit{Monad}\ \mathit{Maybe}\ \texttt{where} \\
\quad \mathit{return} \quad = \mathit{Just} \\
\quad m \gg\!= k = \texttt{case}\ m\ \texttt{of} \\
\qquad\qquad\qquad \mathit{Just}\ x \quad \to k\ x \\
\qquad\qquad\qquad \mathit{Nothing} \to \mathit{Nothing}
\end{array}$$

$$\begin{array}{l}
\texttt{instance}\ \mathit{Functor}\ \mathit{Maybe}\ \texttt{where} \\
\quad \mathit{fmap}\ f\ \ \mathit{Nothing} = \mathit{Nothing} \\
\quad \mathit{fmap}\ f\ (\mathit{Just}\ a) = \mathit{Just}\ (f\ a)
\end{array}$$

$$\begin{array}{l}
\texttt{instance}\ \mathit{Applicative}\ \mathit{Maybe}\ \texttt{where} \\
\quad \mathit{pure} \qquad\quad = \mathit{return} \\
\quad m1 \texttt{<*>} m2 = m1 \gg\!= (\lambda f \to m2 \gg\!= (\lambda x \to \mathit{return}\ (f\ x)))
\end{array}$$

# Leyes de mónadas

$$
\begin{array}{rcl}
\mathit{return}\ x \gg= f & = & f\ x \\
m \gg= \mathit{return} & = & m \\
(m \gg= f) \gg= g & = & m \gg= \lambda x \rightarrow (f\ x \gg= g)
\end{array}
$$

# Composición de funciones monádicas

Composición de Kleisli.

$$(\gg\!\!=\!\!>) :: Monad\ m \Rightarrow (a \to m\ b) \to (b \to m\ c) \to a \to m\ c$$
$$f \gg\!\!=\!\!> g = \lambda a \to f\ a \gg\!= g$$

Propiedades:

$$\begin{aligned} return \gg\!\!=\!\!> f &= f \\ f \gg\!\!=\!\!> return &= f \\ f \gg\!\!=\!\!> (g \gg\!\!=\!\!> h) &= (f \gg\!\!=\!\!> g) \gg\!\!=\!\!> h \end{aligned}$$

Se prueban fácilmente usando las leyes de mónadas.

# Notación do

$$
\begin{aligned}
\mathtt{do}\ m &= m \\
\mathtt{do}\ \{x \leftarrow m; m'\} &= m \ggg\!= \lambda x \rightarrow \mathtt{do}\ m' \\
\mathtt{do}\ \{m; m'\} &= m \gg \mathtt{do}\ m'
\end{aligned}
$$

# Evaluador con Fallas (notación do)

$$
\begin{array}{l}
eval :: Expr \rightarrow Maybe\ Int \\
eval\ (Val\ n) \quad\ \ = return\ n \\
eval\ (Add\ x\ y) = \mathtt{do}\ a \leftarrow eval\ x \\
\qquad\qquad\qquad\qquad\quad b \leftarrow eval\ y \\
\qquad\qquad\qquad\qquad\quad return\ (a + b) \\
eval\ (Div\ x\ y)\ \ = \mathtt{do}\ a \leftarrow eval\ x \\
\qquad\qquad\qquad\qquad\quad b \leftarrow eval\ y \\
\qquad\qquad\qquad\qquad\quad a\ `divM`\ b
\end{array}
$$

# Mónada Either

$$\texttt{data}\ \mathit{Either}\ a\ b = \mathit{Left}\ a \mid \mathit{Right}\ b$$

$$\begin{array}{l}
\texttt{instance}\ \mathit{Monad}\ (\mathit{Either}\ e)\ \texttt{where} \\
\quad \begin{array}{lcl}
\mathit{return} & = & \mathit{Right} \\
\mathit{Left}\ \ e \gg\!= _ & = & \mathit{Left}\ e \\
\mathit{Right}\ a \gg\!= f & = & f\ a
\end{array}
\end{array}$$

# Mónada Either

$$\mathtt{data}\ \mathit{Either}\ a\ b = \mathit{Left}\ a \mid \mathit{Right}\ b$$

$$\begin{array}{l}
\mathtt{instance}\ \mathit{Monad}\ (\mathit{Either}\ e)\ \mathtt{where} \\
\quad \mathit{return} \qquad\qquad = \mathit{Right} \\
\quad \mathit{Left}\ \ e \gg\!= _ = \mathit{Left}\ e \\
\quad \mathit{Right}\ a \gg\!= f = f\ a
\end{array}$$

Se corresponde con:

$$\begin{array}{l}
\mathtt{instance}\ \mathit{Applicative}\ (\mathit{Either}\ e)\ \mathtt{where} \\
\quad \mathit{pure} = \mathit{Right} \\
\quad \mathit{Right}\ f \ \texttt{<*>}\ \mathit{Right}\ a = \mathit{Right}\ (f\ a) \\
\quad \mathit{Right}\ f \ \texttt{<*>}\ \mathit{Left}\ e \quad = \mathit{Left}\ e \\
\quad \mathit{Left}\ e \ \ \texttt{<*>}\ _ \qquad\quad = \mathit{Left}\ e
\end{array}$$

# Mónada Either

```
data Either a b = Left a | Right b

instance Monad (Either e) where
  return          = Right
  Left  e >>= _   = Left e
  Right a >>= f   = f a
```

Pero no con:

```
instance Monoid e => Applicative (Either e) where
  pure = Right
  Right f <*> Right a  = Right (f a)
  Left e  <*> Right _  = Left e
  Right _ <*> Left e   = Left e
  Left e  <*> Left e'  = Left (e `mappend` e')
```

# Mónada Either

$$\mathtt{data}\ Either\ a\ b = Left\ a \mid Right\ b$$

$$\begin{array}{l}
\mathtt{instance}\ Monad\ (Either\ e)\ \mathtt{where} \\
\quad return \qquad\quad = Right \\
\quad Left\ \ e \gg\!= _ = Left\ e \\
\quad Right\ a \gg\!= f\ = f\ a
\end{array}$$

Pero no con:

$$\begin{array}{l}
\mathtt{instance}\ Monoid\ e \Rightarrow Applicative\ (Either\ e)\ \mathtt{where} \\
\quad pure = Right \\
\quad Right\ f \ \texttt{<*>}\ Right\ a = Right\ (f\ a) \\
\quad Left\ e \ \ \texttt{<*>}\ Right\ _ = Left\ e \\
\quad Right\ _ \ \texttt{<*>}\ Left\ e \ \ = Left\ e \\
\quad Left\ e \ \ \texttt{<*>}\ Left\ e' \ \ = Left\ (e\ `mappend`\ e')
\end{array}$$

$$\begin{array}{l}
Left\ a \ \texttt{<*>}\ Left\ b = Left\ (a\ `mappend`\ b) \\
Left\ a \gg\!= (\lambda f \to Left\ b \gg\!= (\lambda x \to return\ (f\ x))) = Left\ a
\end{array}$$

# Applicative no monádico

La instancia anterior de *Applicative* para *Either* **no** es una mónada.

$$
\begin{array}{l}
\mathtt{instance}\ \mathit{Monoid}\ e \Rightarrow \mathit{Monad}\ (\mathit{Either}\ e)\ \mathtt{where} \\
\quad \mathit{return} = \mathit{Right} \\
\quad \mathit{Left}\ e \gg\!= f\ = ?? \\
\quad \ldots
\end{array}
$$

# Applicative no monádico

La instancia anterior de *Applicative* para *Either* **no** es una mónada.

$$
\begin{array}{l}
\mathbf{instance}\ \mathit{Monoid}\ e \Rightarrow \mathit{Monad}\ (\mathit{Either}\ e)\ \mathbf{where} \\
\quad \mathit{return} = \mathit{Right} \\
\quad \mathit{Left}\ e \gg\!\!= f = ?? \\
\quad \ldots
\end{array}
$$

- No podemos aplicar f en este caso porque sólo se aplica cuando la primera computación retorna un valor ($\mathit{Right}\ a$).
- Esto no ocurre en la instancia de *Applicative*.

# Diferencia entre functores aplicativos y mónadas

La diferencia entre mónadas y functores aplicativos se puede apreciar en los siguientes operadores condicionales:

$$
\begin{aligned}
&\mathit{ifM} :: \mathit{Monad}\ m \Rightarrow m\ \mathit{Bool} \to m\ a \to m\ a \to m\ a \\
&\mathit{ifM}\ \mathit{mb}\ \mathit{mt}\ \mathit{me} = \mathtt{do}\ b \leftarrow \mathit{mb} \\
&\qquad\qquad\qquad\qquad\quad \mathtt{if}\ b\ \mathtt{then}\ \mathit{mt}\ \mathtt{else}\ \mathit{me}
\end{aligned}
$$

No todas computaciones se ejecutan (se elije entre $mt$ y $me$)

# Diferencia entre functores aplicativos y mónadas

La diferencia entre mónadas y functores aplicativos se puede apreciar en los siguientes operadores condicionales:

$$
\begin{aligned}
&\mathit{ifM} :: \mathit{Monad}\ m \Rightarrow m\ \mathit{Bool} \rightarrow m\ a \rightarrow m\ a \rightarrow m\ a\\
&\mathit{ifM}\ \mathit{mb}\ \mathit{mt}\ \mathit{me} = \mathbf{do}\ b \leftarrow \mathit{mb}\\
&\qquad\qquad\qquad\qquad\quad \mathbf{if}\ b\ \mathbf{then}\ \mathit{mt}\ \mathbf{else}\ \mathit{me}
\end{aligned}
$$

No todas computaciones se ejecutan (se elije entre *mt* y *me*)

$$
\begin{aligned}
&\mathit{ifA} :: \mathit{Applicative}\ f \Rightarrow f\ \mathit{Bool} \rightarrow f\ a \rightarrow f\ a \rightarrow f\ a\\
&\mathit{ifA}\ \mathit{fb}\ \mathit{ft}\ \mathit{fe} = \mathit{cond} \mathbin{<\$>} \mathit{fb} \mathbin{<*>} \mathit{ft} \mathbin{<*>} \mathit{fe}\\
&\quad \mathbf{where}\\
&\qquad \mathit{cond}\ b\ t\ e = \mathbf{if}\ b\ \mathbf{then}\ t\ \mathbf{else}\ e
\end{aligned}
$$

Las tres computaciones (*fb*, *ft* y *fe*) se ejecutan y finalmente se elije uno de los resultados.

# Mónada de estado

$$\texttt{newtype}\ State\ s\ a = State\ (s \to (a, s))$$

$$\begin{array}{l} runState :: State\ s\ a \to (s \to (a, s)) \\ runState\ (State\ f) = f \end{array}$$

$$\begin{array}{l} \texttt{instance}\ Monad\ (State\ s)\ \texttt{where} \\ \quad return\ a = State\ \$\ \lambda s \to (a, s) \\ \quad m \gg\!= f = State\ \$\ \lambda s \to \texttt{let}\ (a, s') = runState\ m\ s \\ \qquad\qquad\qquad\qquad\qquad\quad \texttt{in}\ \ runState\ (f\ a)\ s' \end{array}$$

Forma alternativa de escribir la definición de $(\gg\!=)$:

$$\begin{array}{l} (State\ g) \gg\!= f = State\ \$\ \lambda s \to \texttt{let}\ (a, s') \ \ = g\ s \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad\ \ State\ k = f\ a \\ \qquad\qquad\qquad\qquad\qquad\quad \texttt{in}\ \ k\ s' \end{array}$$

# Funciones sobre estado

$$get :: State\ s\ s$$
$$get = State\ \$\ \lambda s \rightarrow (s, s)$$

$$put :: s \rightarrow State\ s\ ()$$
$$put\ s = State\ \$\ \lambda\_ \rightarrow ((), s)$$

$$modify :: (s \rightarrow s) \rightarrow State\ s\ ()$$
$$modify\ f = get \gg\!= \lambda s \rightarrow put\ (f\ s)$$

$$evalState :: State\ s\ a \rightarrow s \rightarrow a$$
$$evalState\ m\ s = fst\ \$\ runState\ m\ s$$

$$execState :: State\ s\ a \rightarrow s \rightarrow s$$
$$execState\ m\ s = snd\ \$\ runState\ m\ s$$

# Ejemplo: contar número de sumas en una expresión

$$
\begin{aligned}
&\mathit{tick} :: \mathit{State}\ \mathit{Int}\ ()\\
&\mathit{tick} = \mathit{modify}\ (+1)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{evalS} :: \mathit{Expr} \rightarrow \mathit{State}\ \mathit{Int}\ \mathit{Int}\\
&\mathit{evalS}\ (\mathit{Val}\ n) = \mathit{return}\ n\\
&\mathit{evalS}\ (\mathit{Add}\ e\ e') = \mathtt{do}\ a \leftarrow \mathit{evalS}\ e\\
&\qquad\qquad\qquad\qquad\quad\ b \leftarrow \mathit{evalS}\ e'\\
&\qquad\qquad\qquad\qquad\quad\ \mathit{tick}\\
&\qquad\qquad\qquad\qquad\quad\ \mathit{return}\ (a + b)
\end{aligned}
$$

$$\mathit{nroSumas}\ e = \mathit{execState}\ (\mathit{evalS}\ e)\ 0$$

# Evaluador con Variables

$$
\begin{aligned}
\texttt{data}\ \mathit{Expr} = &\ \mathit{Val\ Int}\\
|\ &\ \mathit{Add\ Expr\ Expr}\\
|\ &\ \mathit{Var\ ID} \quad \text{-- variables}\\
|\ &\ \mathit{Assign\ ID\ Expr} \quad \text{-- asignación}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{eval} :: \mathit{Expr} \to \mathit{State}\ (\mathit{Map\ ID\ Int})\ \mathit{Int}\\
&\mathit{eval}\ (\mathit{Val}\ n) = \mathit{return}\ n\\
&\mathit{eval}\ (\mathit{Add}\ e\ e') = \texttt{do}\ a \leftarrow \mathit{eval}\ e\\
&\qquad\qquad\qquad\quad\ b \leftarrow \mathit{eval}\ e'\\
&\qquad\qquad\qquad\quad\ \mathit{return}\ (a + b)\\
&\mathit{eval}\ (\mathit{Var}\ v) = \texttt{do}\ s \leftarrow \mathit{get}\\
&\qquad\qquad\qquad\quad\ \mathit{return}\ (\mathit{fromJust}\ \$\ \mathit{lookup}\ v\ s)\\
&\mathit{eval}\ (\mathit{Assign}\ v\ e) = \texttt{do}\ a \leftarrow \mathit{eval}\ e\\
&\qquad\qquad\qquad\quad\ s \leftarrow \mathit{get}\\
&\qquad\qquad\qquad\quad\ \mathit{put}\ (\mathit{insert}\ v\ a\ s)\\
&\qquad\qquad\qquad\quad\ \mathit{return}\ a
\end{aligned}
$$

# Mónada de Estado

$$
\begin{array}{l}
\mathtt{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadState}\ s\ m \mid m \rightarrow s\ \mathtt{where} \\
\quad \mathit{get} :: m\ s \\
\quad \mathit{put} :: s \rightarrow m\ ()
\end{array}
$$

$$
\begin{array}{l}
\mathit{modify} :: \mathit{MonadState}\ s\ m \Rightarrow (s \rightarrow s) \rightarrow m\ () \\
\mathit{modify}\ f = \mathtt{do}\ s \leftarrow \mathit{get} \\
\qquad\qquad\qquad\ \ \mathit{put}\ (f\ s)
\end{array}
$$

# Mónada de Estado

$$\mathtt{class}\ Monad\ m \Rightarrow MonadState\ s\ m \mid m \rightarrow s\ \mathtt{where}$$
$$\quad get :: m\ s$$
$$\quad put :: s \rightarrow m\ ()$$

$$modify :: MonadState\ s\ m \Rightarrow (s \rightarrow s) \rightarrow m\ ()$$
$$modify\ f = \mathtt{do}\ s \leftarrow get$$
$$\qquad\qquad\quad put\ (f\ s)$$

$$\mathtt{instance}\ MonadState\ s\ (State\ s)\ \mathtt{where}$$
$$\quad get \quad = State\ \$\ \lambda s \rightarrow (s, s)$$
$$\quad put\ s = State\ \$\ \lambda\_ \rightarrow ((), s)$$

# Mónada List

$$
\begin{array}{l}
\texttt{instance}\ \mathit{Monad}\ [\,]\ \texttt{where} \\
\quad \mathit{return}\ x = [x] \\
\quad \mathit{xs} \gg\!= f = [y \mid x \leftarrow \mathit{xs}, y \leftarrow f\ x] \\
\qquad\qquad\qquad \text{-- concat (map f xs)}
\end{array}
$$

**Ejemplo:** Suma de todos los pares de valores de dos listas

$$
\begin{array}{l}
\mathit{sumnd} :: \mathit{Num}\ a \Rightarrow [a] \rightarrow [a] \rightarrow [a] \\
\mathit{sumnd}\ \mathit{xs}\ \mathit{ys} = \texttt{do}\ x \leftarrow \mathit{xs} \\
\qquad\qquad\qquad\quad\ y \leftarrow \mathit{ys} \\
\qquad\qquad\qquad\quad\ \mathit{return}\ (x + y)
\end{array}
$$

$$
\begin{array}{l}
> \mathit{sumnd}\ [1,3]\ [4,7] \\
[5,8,7,10]
\end{array}
$$

# 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:

$$
\begin{array}{l}
\mathtt{class}\ \mathit{Expr}\ e\ \mathtt{where} \\
\quad \mathit{val} :: \mathit{Int} \rightarrow e \\
\quad \mathit{add} :: e \rightarrow e \rightarrow e
\end{array}
$$

$$
\begin{array}{l}
\mathit{expr1} :: \mathit{Expr}\ e \Rightarrow e \\
\mathit{expr1} = \mathit{add}\ (\mathit{val}\ 8)\ (\mathit{add}\ (\mathit{add}\ (\mathit{val}\ 2)\ (\mathit{val}\ 1))\ (\mathit{val}\ 4))
\end{array}
$$

# Intérpretes

Defino las interpretaciones como instancias de la clase

# Intérpretes

Defino las interpretaciones como instancias de la clase

Evaluación:

$$\mathtt{data}\ \mathit{Eval} = \mathit{E}\ \mathit{Int}$$

$$\begin{array}{l}
\mathtt{instance}\ \mathit{Expr}\ \mathit{Eval}\ \mathtt{where} \\
\quad \mathit{val}\ x \qquad\qquad\quad\;\; = \mathit{E}\ x \\
\quad \mathit{add}\ (\mathit{E}\ x)\ (\mathit{E}\ y) = \mathit{E}\ (x + y)
\end{array}$$

# 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:

$$\texttt{data}\ \mathit{Eval} = E\ \mathit{Int}$$

$$\begin{array}{l}
\texttt{instance}\ \mathit{Expr}\ \mathit{Eval}\ \texttt{where} \\
\quad \mathit{val}\ x \qquad\qquad\quad = E\ x \\
\quad \mathit{add}\ (E\ x)\ (E\ y) = E\ (x + y)
\end{array}$$

Pretty-printing:

$$\texttt{data}\ \mathit{PP} = P\ \mathit{String}$$

$$\begin{array}{l}
\texttt{instance}\ \mathit{Expr}\ \mathit{PP}\ \texttt{where} \\
\quad \mathit{val}\ x \qquad\qquad\quad = P\ (\mathit{show}\ x) \\
\quad \mathit{add}\ (P\ x)\ (P\ y) = P\ (\texttt{"("} + \!\!+\ x + \!\!+\ \texttt{" + "} + \!\!+\ y + \!\!+\ \texttt{")"})
\end{array}$$

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

$$\textbf{class}\ \mathit{ExprMult}\ e\ \textbf{where}$$
$$\quad \mathit{mult} :: e \to e \to e$$

$$\mathit{expr2} :: (\mathit{Expr}\ e, \mathit{ExprMult}\ e) \Rightarrow e$$
$$\mathit{expr2} = \mathit{add}\ (\mathit{mult}\ \mathit{expr1}\ (\mathit{val}\ 4))\ (\mathit{val}\ 2)$$

# 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

$$
\begin{array}{l}
\texttt{class}\ \mathit{Expr}\ e\ \texttt{where} \\
\quad \mathit{val} \quad\ \, :: \mathit{Int} \rightarrow e \\
\quad \mathit{add} \quad\ :: e \rightarrow e \rightarrow e \\
\quad \mathit{isZero} :: e \rightarrow e \\
\quad \mathit{ifE} \quad\ \ :: e \rightarrow e \rightarrow e \rightarrow e
\end{array}
$$

# 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
```

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:

```haskell
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:

$$
\begin{array}{l}
\texttt{class}\ \mathit{TExpr}\ e\ \texttt{where} \\
\quad \mathit{valT} \quad\ \ :: \mathit{Int} \rightarrow e\ \mathit{Int} \\
\quad \mathit{addT} \quad\ :: e\ \mathit{Int} \rightarrow e\ \mathit{Int} \rightarrow e\ \mathit{Int} \\
\quad \mathit{isZeroT} :: e\ \mathit{Int} \rightarrow e\ \mathit{Bool} \\
\quad \mathit{ifT} \quad\quad :: e\ \mathit{Bool} \rightarrow e\ t \rightarrow e\ t \rightarrow e\ t
\end{array}
$$

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

$$
\begin{array}{l}
\mathit{exprT} :: \mathit{TExpr}\ e \Rightarrow e\ \mathit{Int} \\
\mathit{exprT} = \mathit{ifT}\ (\mathit{isZeroT}\ (\mathit{valT}\ 2))\ (\mathit{valT}\ 2)\ (\mathit{valT}\ 3)
\end{array}
$$

# 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

$$\mathbf{data}\ \mathit{TEval}\ t = \mathit{TE}\ t$$

$$\begin{array}{ll}
\mathbf{instance}\ \mathit{TExpr}\ \mathit{TEval}\ \mathbf{where} & \\
\quad \mathit{valT}\ x & = \mathit{TE}\ x \\
\quad \mathit{addT}\ (\mathit{TE}\ x)\ (\mathit{TE}\ y) & = \mathit{TE}\ (x + y) \\
\quad \mathit{isZeroT}\ (\mathit{TE}\ x) & = \mathit{TE}\ (x \equiv 0) \\
\quad \mathit{ifT}\ (\mathit{TE}\ c)\ (\mathit{TE}\ x)\ (\mathit{TE}\ y) = \mathit{TE}\ (\mathbf{if}\ c\ \mathbf{then}\ x\ \mathbf{else}\ y) &
\end{array}$$

# 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)
```

# Abordaje Funcional a EDSLs

Alberto Pardo Marcos Viera

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

ECI 2024

# Deep Embedding de Lenguajes Tipados

# Lenguajes Tipados - Deep Embedding

Volviendo al lenguaje con distintos tipos, pero ahora como deep embedding

$$
\begin{array}{l}
\mathtt{data}\ \mathit{Expr} :: * \mathtt{where} \\
\quad \mathit{Val} \quad\ \ :: \mathit{Int} \to \mathit{Expr} \\
\quad \mathit{Add} \quad\ :: \mathit{Expr} \to \mathit{Expr} \to \mathit{Expr} \\
\quad \mathit{IsZero} :: \mathit{Expr} \to \mathit{Expr} \\
\quad \mathit{If} \qquad :: \mathit{Expr} \to \mathit{Expr} \to \mathit{Expr} \to \mathit{Expr}
\end{array}
$$

# Evaluador con type-checking

Evaluador parcial, o con type-checking dinámico

```
data Res = RI Int | RB Bool

eval :: Expr → Res
eval (Val n)      = RI n
eval (Add e1 e2) = case (eval e1, eval e2) of
                     (RI n1, RI n2) → RI (n1 + n2)
                     _              → error "type error: Add"
eval (IsZero e)   = case eval e of
                     RI n  → RB (n ≡ 0)
                     _     → error "type error: IsZero"
eval (If e1 e2 e3) = case eval e1 of
                     RB b →
                       case (eval e2, eval e3) of
                         (RI n2, RI n3)  → RI  $ if b then n2 else n3
                         (RB b2, RB b3) → RB $ if b then b2 else b3
                         _               → error "type error: If branches"
                     _     → error "type error: If condition"
```

# Type-checking

Podría desacoplar el type-checking de la evaluación

$$
\begin{array}{l}
\mathtt{data}\ \mathit{EType} = \mathit{TInt} \mid \mathit{TBool} \\
\\
\mathtt{infix}\ 6\ .{:}{:} \\
\\
(.{:}{:}) :: \mathit{Expr} \to \mathit{EType} \to \mathit{Bool} \\
\begin{array}{llll}
(\mathit{Val}\ n) & .{:}{:}\ \mathit{TInt} & = \mathit{True} & \\
(\mathit{Add}\ e1\ e2) & .{:}{:}\ \mathit{TInt} & = e1\ .{:}{:}\ \mathit{TInt} & \wedge\ e2\ .{:}{:}\ \mathit{TInt} \\
(\mathit{IsZero}\ e) & .{:}{:}\ \mathit{TBool} & = e\ \ .{:}{:}\ \mathit{TInt} & \\
(\mathit{If}\ c\ e1\ e2) & .{:}{:}\ t & = c\ \ .{:}{:}\ \mathit{TBool} & \wedge\ e1\ .{:}{:}\ t \wedge e2\ .{:}{:}\ t \\
\_ & .{:}{:}\ \_ & = \mathit{False} &
\end{array} \\
\\
\mathit{wellTyped} :: \mathit{Expr} \to \mathit{Bool} \\
\mathit{wellTyped}\ e = e\ .{:}{:}\ \mathit{TInt} \vee e\ .{:}{:}\ \mathit{TBool}
\end{array}
$$

# Type-checking

Podría desacoplar el type-checking de la evaluación

$$
\begin{array}{l}
\mathtt{data}\ \mathit{EType} = \mathit{TInt} \mid \mathit{TBool} \\
\\
\mathtt{infix}\ 6\ .{:}{:} \\
\\
(.{:}{:}) :: \mathit{Expr} \to \mathit{EType} \to \mathit{Bool} \\
(\mathit{Val}\ n)\quad .{:}{:}\ \mathit{TInt} \quad = \mathit{True} \\
(\mathit{Add}\ e1\ e2)\ .{:}{:}\ \mathit{TInt} \quad = e1\ .{:}{:}\ \mathit{TInt} \wedge e2\ .{:}{:}\ \mathit{TInt} \\
(\mathit{IsZero}\ e)\quad .{:}{:}\ \mathit{TBool} = e\ .{:}{:}\ \mathit{TInt} \\
(\mathit{If}\ c\ e1\ e2)\ .{:}{:}\ t \quad = c\ .{:}{:}\ \mathit{TBool} \wedge e1\ .{:}{:}\ t \wedge e2\ .{:}{:}\ t \\
\_\quad .{:}{:}\ \_ \quad = \mathit{False} \\
\\
\mathit{wellTyped} :: \mathit{Expr} \to \mathit{Bool} \\
\mathit{wellTyped}\ e = e\ .{:}{:}\ \mathit{TInt} \vee e\ .{:}{:}\ \mathit{TBool}
\end{array}
$$

y sólo evaluar en caso de tipar correctamente

$$
\begin{array}{l}
\mathit{safeEval} :: \mathit{Expr} \to \mathit{Res} \\
\mathit{safeEval}\ e \mid \mathit{wellTyped}\ e = \mathit{eval}\ e \\
\qquad\qquad\ \mid \mathit{otherwise} \quad = \mathit{error}\ \texttt{"type error"}
\end{array}
$$

# Tagless - Deep Embedding (Initial)

Podemos usar el enfoque tagless para que el sistema de tipos del lenguaje anfitrión realice el type-checking

$$
\begin{array}{l}
\texttt{class}\ \mathit{ExprT}\ (e :: * \rightarrow *)\ \texttt{where} \\
\quad \mathit{valT} \quad\ \ :: \mathit{Int} \rightarrow e\ \mathit{Int} \\
\quad \mathit{addT} \quad\ :: e\ \mathit{Int} \rightarrow e\ \mathit{Int} \rightarrow e\ \mathit{Int} \\
\quad \mathit{isZeroT} :: e\ \mathit{Int} \rightarrow e\ \mathit{Bool} \\
\quad \mathit{ifT} \qquad :: e\ \mathit{Bool} \rightarrow e\ t \rightarrow e\ t \rightarrow e\ t
\end{array}
$$

# Tagless - Deep Embedding (Initial)

Podemos usar el enfoque tagless para que el sistema de tipos del lenguaje anfitrión realice el type-checking

```
class ExprT (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
```

Necesito empaquetar *Expr* en un $* \to *$ con un phantom type

```
newtype WExpr a = E Expr

instance ExprT WExpr where
  valT n                    = E (Val n)
  addT (E e1) (E e2)        = E (Add e1 e2)
  isZeroT (E e)             = E (IsZero e)
  ifT (E c) (E e1) (E e2)   = E (If c e1 e2)
```

# Intérpretes Tipados

Puedo definir un observador como pretty-printing sin problemas

$$
\begin{array}{ll}
\mathit{ppExpr}\ (E\ (\mathit{Val}\ n)) & = \mathit{show}\ n \\
\mathit{ppExpr}\ (E\ (\mathit{Add}\ e1\ e2)) & = \texttt{"("} \mathbin{+\!\!+} \mathit{ppExpr}\ (E\ e1) \mathbin{+\!\!+} \texttt{" + "} \mathbin{+\!\!+} \mathit{ppExpr}\ (E\ e2) \mathbin{+\!\!+} \texttt{")"} \\
\mathit{ppExpr}\ (E\ (\mathit{IsZero}\ e)) & = \texttt{"isZero("} \mathbin{+\!\!+} \mathit{ppExpr}\ (E\ e) \mathbin{+\!\!+} \texttt{")"} \\
\mathit{ppExpr}\ (E\ (\mathit{If}\ c\ e1\ e2)) & = \texttt{"if "} \mathbin{+\!\!+} \mathit{ppExpr}\ (E\ c) \mathbin{+\!\!+} \\
& \quad \texttt{" then "} \mathbin{+\!\!+} \mathit{ppExpr}\ (E\ e1) \mathbin{+\!\!+} \\
& \quad \texttt{" else "} \mathbin{+\!\!+} \mathit{ppExpr}\ (E\ e2)
\end{array}
$$

# Intérpretes Tipados

Puedo definir un observador como pretty-printing sin problemas

$$
\begin{aligned}
&\mathit{ppExpr}\ (E\ (\mathit{Val}\ n)) &&= \mathit{show}\ n\\
&\mathit{ppExpr}\ (E\ (\mathit{Add}\ e1\ e2)) &&= \texttt{"("} +\!\!+\ \mathit{ppExpr}\ (E\ e1) +\!\!+ \texttt{" + "} +\!\!+\ \mathit{ppExpr}\ (E\ e2) +\!\!+ \texttt{")"}\\
&\mathit{ppExpr}\ (E\ (\mathit{IsZero}\ e)) &&= \texttt{"isZero("} +\!\!+\ \mathit{ppExpr}\ (E\ e) +\!\!+ \texttt{")"}\\
&\mathit{ppExpr}\ (E\ (\mathit{If}\ c\ e1\ e2)) &&= \texttt{"if "} +\!\!+\ \mathit{ppExpr}\ (E\ c) +\!\!+\\
& && \quad \texttt{" then "} +\!\!+\ \mathit{ppExpr}\ (E\ e1) +\!\!+\\
& && \quad \texttt{" else "} +\!\!+\ \mathit{ppExpr}\ (E\ e2)
\end{aligned}
$$

Pero no puedo definir algo como

$$
\begin{aligned}
&\mathit{evalExpr}\ (E\ (\mathit{Val}\ n)) &&= n\\
&\mathit{evalExpr}\ (E\ (\mathit{IsZero}\ e)) &&= \mathit{evalExpr}\ (E\ e) \equiv 0\\
&\mathit{evalExpr}\ (E\ (\mathit{Add}\ e1\ e2)) &&= \mathit{evalExpr}\ (E\ e1) + \mathit{evalExpr}\ (E\ e2)\\
&\mathit{evalExpr}\ (E\ (\mathit{If}\ c\ e1\ e2)) &&= \texttt{if}\ \mathit{evalExpr}\ (E\ c)\ \texttt{then}\ \mathit{evalExpr}\ (E\ e1)\\
& && \qquad\qquad\qquad\qquad\ \ \texttt{else}\ \mathit{evalExpr}\ (E\ e2)
\end{aligned}
$$

# Intérpretes Tipados

Puedo definir un observador como pretty-printing sin problemas

$$
\begin{array}{ll}
\mathit{ppExpr}\ (E\ (\mathit{Val}\ n)) & = \mathit{show}\ n \\
\mathit{ppExpr}\ (E\ (\mathit{Add}\ e1\ e2)) & = \texttt{"("} + \mathit{ppExpr}\ (E\ e1) + \texttt{" + "} + \mathit{ppExpr}\ (E\ e2) + \texttt{")"} \\
\mathit{ppExpr}\ (E\ (\mathit{IsZero}\ e)) & = \texttt{"isZero("} + \mathit{ppExpr}\ (E\ e) + \texttt{")"} \\
\mathit{ppExpr}\ (E\ (\mathit{If}\ c\ e1\ e2)) & = \texttt{"if "} + \mathit{ppExpr}\ (E\ c) + \\
& \quad \texttt{" then "} + \mathit{ppExpr}\ (E\ e1) + \\
& \quad \texttt{" else "} + \mathit{ppExpr}\ (E\ e2)
\end{array}
$$

Pero no puedo definir algo como

$$
\begin{array}{ll}
\mathit{evalExpr}\ (E\ (\mathit{Val}\ n)) & = n \\
\mathit{evalExpr}\ (E\ (\mathit{IsZero}\ e)) & = \mathit{evalExpr}\ (E\ e) \equiv 0 \\
\mathit{evalExpr}\ (E\ (\mathit{Add}\ e1\ e2)) & = \mathit{evalExpr}\ (E\ e1) + \mathit{evalExpr}\ (E\ e2) \\
\mathit{evalExpr}\ (E\ (\mathit{If}\ c\ e1\ e2)) & = \texttt{if}\ \mathit{evalExpr}\ (E\ c)\ \texttt{then}\ \mathit{evalExpr}\ (E\ e1) \\
& \qquad \texttt{else}\ \mathit{evalExpr}\ (E\ e2)
\end{array}
$$

```
Couldn't match expected type 'Int' with actual type 'Bool'
In the expression: evalExpr (E e) == 0
...
```

# Intérpretes Tipados (2)

Tampoco puedo definir

$$
\begin{array}{lcl}
\mathit{evalExpr} :: \mathit{WExpr}\ t \rightarrow t \\
\mathit{evalExpr}\ (E\ (\mathit{Val}\ n)) & = & n \\
\mathit{evalExpr}\ (E\ (\mathit{IsZero}\ e)) & = & \mathit{evalExpr}\ (E\ e) \equiv 0 \\
\mathit{evalExpr}\ (E\ (\mathit{Add}\ e1\ e2)) & = & \mathit{evalExpr}\ (E\ e1) + \mathit{evalExpr}\ (E\ e2) \\
\mathit{evalExpr}\ (E\ (\mathit{If}\ c\ e1\ e2)) & = & \mathtt{if}\ \mathit{evalExpr}\ (E\ c)\ \mathtt{then}\ \mathit{evalExpr}\ (E\ e1) \\
 & & \qquad\qquad\qquad\quad\ \mathtt{else}\ \mathit{evalExpr}\ (E\ e2)
\end{array}
$$

# Intérpretes Tipados (2)

Tampoco puedo definir

$$
\begin{aligned}
&\mathit{evalExpr} :: \mathit{WExpr}\ t \to t\\
&\mathit{evalExpr}\ (E\ (\mathit{Val}\ n)) &&= n\\
&\mathit{evalExpr}\ (E\ (\mathit{IsZero}\ e)) &&= \mathit{evalExpr}\ (E\ e) \equiv 0\\
&\mathit{evalExpr}\ (E\ (\mathit{Add}\ e1\ e2)) &&= \mathit{evalExpr}\ (E\ e1) + \mathit{evalExpr}\ (E\ e2)\\
&\mathit{evalExpr}\ (E\ (\mathit{If}\ c\ e1\ e2)) &&= \mathbf{if}\ \mathit{evalExpr}\ (E\ c)\ \mathbf{then}\ \mathit{evalExpr}\ (E\ e1)\\
& && \phantom{= \mathbf{if}\ \mathit{evalExpr}\ (E\ c)\ }\mathbf{else}\ \mathit{evalExpr}\ (E\ e2)
\end{aligned}
$$

```
Couldn't match expected type 't' with actual type 'Int'
't' is a rigid type variable bound by
  the type signature for:
    evalExpr :: forall t. WExpr t -> t
...
In the expression: n
  In an equation for 'evalExpr': evalExpr (E (Val n)) = n
...
```

# Intérpretes Tipados (3)

Se puede definir un *workaround* usando type-clases

```
class Eval t where
  weval :: WExpr t → t

instance Eval Int where
  weval (E (Val n))      = n
  weval (E (Add e1 e2)) = weval (E e1) + weval (E e2)
  weval (E (If c e1 e2)) = if weval (E c) then weval (E e1) else weval (E e2)

instance Eval Bool where
  weval (E (IsZero e))   = (weval (E e) :: Int) ≡ 0
  weval (E (If c e1 e2)) = if weval (E c) then weval (E e1) else weval (E e2)
```

# Abordaje Funcional a EDSLs

Alberto Pardo Marcos Viera

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

ECI 2024

Tipos de Datos Algebraicos Generalizados (GADTs)

# Tipo de Datos Algebraico

$$\texttt{data}\ Tree\ a = Leaf\ a$$
$$\quad\quad\quad\quad\mid Node\ (Tree\ a)\ (Tree\ a)$$

o alternativamente,

$$\begin{array}{l} \texttt{data}\ Tree :: * \rightarrow *\ \texttt{where} \\ \quad Leaf\ \ :: Tree\ a \\ \quad Node :: Tree\ a \rightarrow a \rightarrow Tree\ a \rightarrow Tree\ a \end{array}$$

introduce:

- un nuevo tipo de datos *Tree* de kind $* \rightarrow *$
- Constructores *Leaf* y *Node*
- la posibilidad de usar los constructores en patterns

# Restricciones

Los constructores de un tipo de datos $T$ deben:

- resultar en el tipo $T$
- resultar en un tipo simple
  - $T\ a_1 \ldots a_n$ con $a_1, \ldots, a_n$ variables de tipo distintas

# Restricciones

Los constructores de un tipo de datos $T$ deben:

- resultar en el tipo $T$
- resultar en un tipo simple
  - $T\ a_1\ ...\ a_n$ con $a_1, ..., a_n$ variables de tipo distintas

Vamos a levantar alguna de estas restricciones.

# Deep embedding untyped

$$
\begin{array}{l}
\mathtt{data}\ \mathit{Expr} :: * \ \mathtt{where} \\
\quad \mathit{Val} \quad\ \ :: \mathit{Int} \to \mathit{Expr} \\
\quad \mathit{Add} \quad\ :: \mathit{Expr} \to \mathit{Expr} \to \mathit{Expr} \\
\quad \mathit{IsZero} :: \mathit{Expr} \to \mathit{Expr} \\
\quad \mathit{If} \qquad :: \mathit{Expr} \to \mathit{Expr} \to \mathit{Expr} \to \mathit{Expr}
\end{array}
$$

Por ejemplo, podemos escribir:

$$\mathit{If}\ (\mathit{IsZero}\ (\mathit{Add}\ (\mathit{Int}\ 0)\ (\mathit{Int}\ 1)))\ (\mathit{Val}\ 3)\ (\mathit{Val}\ 4)$$

que representa la sintaxis abstracta de la siguiente expresión escrita en una sintaxis concreta:

$$\mathtt{if}\ \mathit{isZero}\ (0 + 1)\ \mathtt{then}\ 3\ \mathtt{else}\ 4$$

Pero podemos escribir también térmimos mal tipados de acuerdo al sistema de tipos del EDSL.

# Deep embedding tipado

La idea es codificar el tipo del término que se representa en el propio tipo Haskell.

$$
\begin{array}{l}
\texttt{data}\ \mathit{Expr} :: * \ \texttt{where} \\
\quad \mathit{Val} \quad\ :: \mathit{Int} \to \mathit{Expr} \\
\quad \mathit{Add} \quad :: \mathit{Expr} \to \mathit{Expr} \to \mathit{Expr} \\
\quad \mathit{IsZero} :: \mathit{Expr} \to \mathit{Expr} \\
\quad \mathit{If} \qquad :: \mathit{Expr} \to \mathit{Expr} \to \mathit{Expr} \to \mathit{Expr}
\end{array}
$$

# Deep embedding tipado

La idea es codificar el tipo del término que se representa en el propio tipo Haskell.

```
data Expr :: * where
  Val    :: Int → Expr
  Add    :: Expr → Expr → Expr
  IsZero :: Expr → Expr
  If     :: Expr → Expr → Expr → Expr
```

```
data Expr :: * → * where
  Val    :: Int → Expr Int
  Add    :: Expr Int → Expr Int → Expr Int
  IsZero :: Expr Int → Expr Bool
  If     :: Expr Bool → Expr t → Expr t → Expr t
```

# GADTs

Los GADTs levantan la restricción de que los constructores deben resultar en un tipo simple.

- Los constructores pueden resultar en un subconjunto del tipo
- Consecuencias interesantes en el pattern matching
  - cuando se analiza un *Expr Int*, éste no puede ser construido por *IsZero*
  - cuando se analiza un *Expr Bool*, éste no puede ser construido por *Val* o *Add*
  - cuando se analiza un *Expr Bool*, si encontramos *IsZero* en el pattern, sabemos que tenemos un *Expr Bool*
  - etc

# Evaluación usando GADTs: evaluador tagless

$$
\begin{aligned}
&\mathit{eval} :: \mathit{Expr}\ t \to t\\
&\mathit{eval}\ (\mathit{Val}\ n) = n\\
&\mathit{eval}\ (\mathit{Add}\ e1\ e2) = \mathit{eval}\ e1 + \mathit{eval}\ e2\\
&\mathit{eval}\ (\mathit{IsZero}\ e) = \mathit{eval}\ e \equiv 0\\
&\mathit{eval}\ (\mathit{If}\ c\ e1\ e2) = \mathtt{if}\ \mathit{eval}\ c\ \mathtt{then}\ \mathit{eval}\ e1\ \mathtt{else}\ \mathit{eval}\ e2
\end{aligned}
$$

# Evaluación usando GADTs: evaluador tagless

$$
\begin{aligned}
&\mathit{eval} :: \mathit{Expr}\ t \to t \\
&\mathit{eval}\ (\mathit{Val}\ n) &&= n \\
&\mathit{eval}\ (\mathit{Add}\ e1\ e2) &&= \mathit{eval}\ e1 + \mathit{eval}\ e2 \\
&\mathit{eval}\ (\mathit{IsZero}\ e) &&= \mathit{eval}\ e \equiv 0 \\
&\mathit{eval}\ (\mathit{If}\ c\ e1\ e2) &&= \mathtt{if}\ \mathit{eval}\ c\ \mathtt{then}\ \mathit{eval}\ e1\ \mathtt{else}\ \mathit{eval}\ e2
\end{aligned}
$$

- No hay posibilidad de fallos en tiempo de ejecución (salvo $\perp$)
- No se requieren tags
- El pattern matching sobre un GADT requiere signatura de tipo

# GADTs incluyen existenciales

Si extendemos el lenguaje con la construcción y proyección de pares:

$$
\begin{array}{l}
\mathtt{data}\ \mathit{Expr} :: * \to * \ \mathtt{where} \\
\quad \begin{array}{ll}
\mathit{Val} & :: \mathit{Int} \to \mathit{Expr}\ \mathit{Int} \\
\mathit{Add} & :: \mathit{Expr}\ \mathit{Int} \to \mathit{Expr}\ \mathit{Int} \to \mathit{Expr}\ \mathit{Int} \\
\mathit{IsZero} & :: \mathit{Expr}\ \mathit{Int} \to \mathit{Expr}\ \mathit{Bool} \\
\mathit{If} & :: \mathit{Expr}\ \mathit{Bool} \to \mathit{Expr}\ t \to \mathit{Expr}\ t \to \mathit{Expr}\ t \\
\mathit{Pair} & :: \mathit{Expr}\ a \to \mathit{Expr}\ b \to \mathit{Expr}\ (a, b) \\
\mathit{Fst} & :: \mathit{Expr}\ (a, b) \to \mathit{Expr}\ a \\
\mathit{Snd} & :: \mathit{Expr}\ (a, b) \to \mathit{Expr}\ b
\end{array}
\end{array}
$$

# GADTs incluyen existenciales

Si extendemos el lenguaje con la construcción y proyección de pares:

$$
\begin{array}{l}
\texttt{data}\ \mathit{Expr} :: * \to * \ \texttt{where} \\
\quad \mathit{Val} \quad :: \mathit{Int} \to \mathit{Expr}\ \mathit{Int} \\
\quad \mathit{Add} \quad :: \mathit{Expr}\ \mathit{Int} \to \mathit{Expr}\ \mathit{Int} \to \mathit{Expr}\ \mathit{Int} \\
\quad \mathit{IsZero} :: \mathit{Expr}\ \mathit{Int} \to \mathit{Expr}\ \mathit{Bool} \\
\quad \mathit{If} \quad :: \mathit{Expr}\ \mathit{Bool} \to \mathit{Expr}\ t \to \mathit{Expr}\ t \to \mathit{Expr}\ t \\
\quad \mathit{Pair} \quad :: \mathit{Expr}\ a \to \mathit{Expr}\ b \to \mathit{Expr}\ (a, b) \\
\quad \mathit{Fst} \quad :: \mathit{Expr}\ (a, b) \to \mathit{Expr}\ a \\
\quad \mathit{Snd} \quad :: \mathit{Expr}\ (a, b) \to \mathit{Expr}\ b
\end{array}
$$

Para *Fst* y *Snd* se esconde el tipo del componente no proyectado

# GADTs incluyen existenciales

Si extendemos el lenguaje con la construcción y proyección de pares:

$$
\begin{array}{l}
\mathtt{data}\ \mathit{Expr} :: * \to * \ \mathtt{where} \\
\quad \begin{array}{ll}
\mathit{Val} & :: \mathit{Int} \to \mathit{Expr}\ \mathit{Int} \\
\mathit{Add} & :: \mathit{Expr}\ \mathit{Int} \to \mathit{Expr}\ \mathit{Int} \to \mathit{Expr}\ \mathit{Int} \\
\mathit{IsZero} & :: \mathit{Expr}\ \mathit{Int} \to \mathit{Expr}\ \mathit{Bool} \\
\mathit{If} & :: \mathit{Expr}\ \mathit{Bool} \to \mathit{Expr}\ t \to \mathit{Expr}\ t \to \mathit{Expr}\ t \\
\mathit{Pair} & :: \mathit{Expr}\ a \to \mathit{Expr}\ b \to \mathit{Expr}\ (a, b) \\
\mathit{Fst} & :: \mathit{Expr}\ (a, b) \to \mathit{Expr}\ a \\
\mathit{Snd} & :: \mathit{Expr}\ (a, b) \to \mathit{Expr}\ b
\end{array}
\end{array}
$$

Para *Fst* y *Snd* se esconde el tipo del componente no proyectado
Es como tener un tipo *existencial*:

$$\mathtt{data}\ \mathit{Expr}\ a = \ldots \mid \forall\ b.\mathit{Fst}\ (\mathit{Expr}\ (a, b))$$

# Ejemplo: Vectores

Un vector es una lista con largo:

$$\begin{array}{l}
\mathtt{data}\ \mathit{Vec}\ a\ n\ \mathtt{where} \\
\quad \mathit{Nil}\ \ \ :: \mathit{Vec}\ a\ \mathit{Zero} \\
\quad \mathit{Cons} :: a \to \mathit{Vec}\ a\ n \to \mathit{Vec}\ a\ (\mathit{Succ}\ n)
\end{array}$$

Los números naturales los vamos a codificar como tipos vacíos:

$$\begin{array}{l}
\mathtt{data}\ \mathit{Zero} \\
\mathtt{data}\ \mathit{Succ}\ a
\end{array}$$

De esta forma, en el tipo del vector tenemos codificado su largo:

$$\begin{array}{ll}
\mathit{Nil} & :: \mathit{Vec}\ \mathit{Int}\ \mathit{Zero} \\
\mathit{Cons}\ 3\ \mathit{Nil} & :: \mathit{Vec}\ \mathit{Int}\ (\mathit{Succ}\ \mathit{Zero}) \\
\mathit{Cons}\ 2\ (\mathit{Cons}\ 3\ \mathit{Nil}) & :: \mathit{Vec}\ \mathit{Int}\ (\mathit{Succ}\ (\mathit{Succ}\ \mathit{Zero}))
\end{array}$$

# head y tail

Las definiciones de *head* y *tail* son ahora seguras:

$$
\begin{aligned}
&head :: Vec\ a\ (Succ\ n) \rightarrow a\\
&head\ (Cons\ x\ xs) = x
\end{aligned}
$$

$$
\begin{aligned}
&tail :: Vec\ a\ (Succ\ n) \rightarrow Vec\ a\ n\\
&tail\ (Cons\ x\ xs) = xs
\end{aligned}
$$

El caso *Nil* es excluído porque no satisface el requerimiento de que la lista de entrada tenga largo mayor que cero.

Por lo tanto, las expresiones:

$$
\begin{aligned}
&head\ Nil\\
&tail\ Nil
\end{aligned}
$$

resultan en un error de tipo.

# Funciones sobre vectores

$$
\begin{aligned}
&map :: (a \rightarrow b) \rightarrow Vec\ a\ n \rightarrow Vec\ b\ n \\
&map\ f\ Nil \qquad\qquad\quad = Nil \\
&map\ f\ (Cons\ x\ xs) = Cons\ (f\ x)\ (map\ f\ xs)
\end{aligned}
$$

# Funciones sobre vectores

$$
\begin{aligned}
&map :: (a \to b) \to Vec\ a\ n \to Vec\ b\ n \\
&map\ f\ Nil \qquad\qquad\quad = Nil \\
&map\ f\ (Cons\ x\ xs) = Cons\ (f\ x)\ (map\ f\ xs)
\end{aligned}
$$

$$
\begin{aligned}
&zipWith :: (a \to b \to c) \to Vec\ a\ n \to Vec\ b\ n \to Vec\ c\ n \\
&zipWith\ f\ Nil \qquad\qquad\ Nil \qquad\qquad\quad = Nil \\
&zipWith\ f\ (Cons\ x\ xs)\ (Cons\ y\ ys) = Cons\ (f\ x\ y) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (zipWith\ f\ xs\ ys)
\end{aligned}
$$

La función *zipWith* requiere que los vectores tengan el mismo largo

# Funciones sobre vectores (2)

$$
\begin{array}{l}
snoc :: Vec\ a\ n \rightarrow a \rightarrow Vec\ a\ (Succ\ n) \\
snoc\ Nil\ \qquad\qquad\quad y = Cons\ y\ Nil \\
snoc\ (Cons\ x\ xs)\ y = Cons\ x\ (snoc\ xs\ y)
\end{array}
$$

# Funciones sobre vectores (2)

$$
\begin{aligned}
&\mathit{snoc} :: \mathit{Vec}\ a\ n \to a \to \mathit{Vec}\ a\ (\mathit{Succ}\ n)\\
&\mathit{snoc}\ \mathit{Nil}\ \qquad\qquad\ y = \mathit{Cons}\ y\ \mathit{Nil}\\
&\mathit{snoc}\ (\mathit{Cons}\ x\ \mathit{xs})\ y = \mathit{Cons}\ x\ (\mathit{snoc}\ \mathit{xs}\ y)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{reverse} :: \mathit{Vec}\ a\ n \to \mathit{Vec}\ a\ n\\
&\mathit{reverse}\ \mathit{Nil}\ \qquad\qquad = \mathit{Nil}\\
&\mathit{reverse}\ (\mathit{Cons}\ x\ \mathit{xs}) = \mathit{snoc}\ (\mathit{reverse}\ \mathit{xs})\ x
\end{aligned}
$$

# Concatenación de vectores

$$(+\!\!+) :: \mathit{Vec}\ a\ m \rightarrow \mathit{Vec}\ a\ n \rightarrow \mathit{Vec}\ a\ (m \oplus n)$$

# Concatenación de vectores

$$(+\!\!+) :: Vec\ a\ m \rightarrow Vec\ a\ n \rightarrow Vec\ a\ (m \oplus n)$$

Podemos calcular $m \oplus n$ de la siguiente manera:

- construir evidencia explícita
- utilizar una type family (función a nivel de tipos)

# Evidencia explícita

Codificar la suma como otro GADT:

$$\begin{array}{l}
\mathtt{data}\ \mathit{Sum}\ m\ n\ s\ \mathtt{where} \\
\quad \mathit{SumZero} :: \mathit{Sum}\ \mathit{Zero}\ n\ n \\
\quad \mathit{SumSucc} :: \mathit{Sum}\ m\ n\ s \to \mathit{Sum}\ (\mathit{Succ}\ m)\ n\ (\mathit{Succ}\ s)
\end{array}$$

$$\begin{array}{llll}
\multicolumn{4}{l}{\mathit{appV} :: \mathit{Sum}\ m\ n\ s \to \mathit{Vec}\ a\ m \to \mathit{Vec}\ a\ n \to \mathit{Vec}\ a\ s} \\
\mathit{appV}\ \mathit{SumZero} & \mathit{Nil} & \mathit{ys} & = \mathit{ys} \\
\mathit{appV}\ (\mathit{SumSucc}\ p) & (\mathit{Cons}\ x\ \mathit{xs}) & \mathit{ys} & = \mathit{Cons}\ x\ (\mathit{appV}\ p\ \mathit{xs}\ \mathit{ys})
\end{array}$$

# Evidencia explícita

Codificar la suma como otro GADT:

$$
\begin{array}{l}
\mathtt{data}\ \mathit{Sum}\ m\ n\ s\ \mathtt{where} \\
\quad \mathit{SumZero} :: \mathit{Sum}\ \mathit{Zero}\ n\ n \\
\quad \mathit{SumSucc} :: \mathit{Sum}\ m\ n\ s \to \mathit{Sum}\ (\mathit{Succ}\ m)\ n\ (\mathit{Succ}\ s)
\end{array}
$$

$$
\begin{array}{llll}
\mathit{appV} :: \mathit{Sum}\ m\ n\ s \to \mathit{Vec}\ a\ m \to \mathit{Vec}\ a\ n \to \mathit{Vec}\ a\ s \\
\mathit{appV}\ \mathit{SumZero} & \mathit{Nil} & \mathit{ys} & = \mathit{ys} \\
\mathit{appV}\ (\mathit{SumSucc}\ p) & (\mathit{Cons}\ x\ \mathit{xs}) & \mathit{ys} & = \mathit{Cons}\ x\ (\mathit{appV}\ p\ \mathit{xs}\ \mathit{ys})
\end{array}
$$

Desventaja: tenemos que construir la evidencia a mano

# Type family

$$
\begin{array}{l}
\texttt{type}\ \mathit{family}\ (m :: *)\ \mathbin{:\!+\!:}\ (n :: *) :: * \\
\texttt{type instance}\ \mathit{Zero}\ \qquad \mathbin{:\!+\!:}\ n = n \\
\texttt{type instance}\ (\mathit{Succ}\ m)\ \mathbin{:\!+\!:}\ n = \mathit{Succ}\ (m\ \mathbin{:\!+\!:}\ n)
\end{array}
$$

$$
\begin{array}{l}
(+\!\!+) :: \mathit{Vec}\ a\ m \rightarrow \mathit{Vec}\ a\ n \rightarrow \mathit{Vec}\ a\ (m\ \mathbin{:\!+\!:}\ n) \\
\mathit{Nil}\ \qquad\quad +\!\!+\ ys = ys \\
\mathit{Cons}\ x\ xs +\!\!+\ ys = \mathit{Cons}\ x\ (xs +\!\!+\ ys)
\end{array}
$$

# Convertir entre listas y vectores

Sin problemas:

$$
\begin{aligned}
&toList :: Vec\ a\ n \rightarrow [a] \\
&toList\ Nil \qquad\qquad = [\,] \\
&toList\ (Cons\ x\ xs) = x : toList\ xs
\end{aligned}
$$

# Convertir entre listas y vectores

Sin problemas:

$$\begin{aligned}
&toList :: Vec\ a\ n \to [a] \\
&toList\ Nil \qquad\qquad = [\,] \\
&toList\ (Cons\ x\ xs) = x : toList\ xs
\end{aligned}$$

No funciona:

$$\begin{aligned}
&fromList :: [a] \to Vec\ a\ n \\
&fromList\ [\,] \qquad\ = Nil \\
&fromList\ (x : xs) = Cons\ x\ (fromList\ xs)
\end{aligned}$$

# Convertir entre listas y vectores

Sin problemas:

$$
\begin{aligned}
&\mathit{toList} :: \mathit{Vec}\ a\ n \to [a] \\
&\mathit{toList}\ \mathit{Nil} \qquad\qquad = [\,] \\
&\mathit{toList}\ (\mathit{Cons}\ x\ \mathit{xs}) = x : \mathit{toList}\ \mathit{xs}
\end{aligned}
$$

No funciona:

$$
\begin{aligned}
&\mathit{fromList} :: [a] \to \mathit{Vec}\ a\ n \\
&\mathit{fromList}\ [\,] \qquad\quad = \mathit{Nil} \\
&\mathit{fromList}\ (x : \mathit{xs}) = \mathit{Cons}\ x\ (\mathit{fromList}\ \mathit{xs})
\end{aligned}
$$

El tipo dice que el resultado tiene que ser polimórfico en $n$, pero no lo es.

# De listas a vectores

Se puede:

- especificar el largo
- esconder el largo usando un tipo existencial

# Especificando el largo

Los números naturales al nivel de los tipos los reflejamos al nivel de los valores usando un tipo **singleton**.

```
data SNat n where
  Zero :: SNat Zero
  Succ :: SNat n → SNat (Succ n)
```

*SNat n* tiene solo un valor por cada *n*:

```
Zero               :: SNat Zero
Succ Zero          :: SNat (Succ Zero)
Succ (Succ Zero)   :: SNat (Succ (Succ Zero))
```

# Especificando el largo

Los números naturales al nivel de los tipos los reflejamos al nivel de los valores usando un tipo **singleton**.

$$
\begin{array}{l}
\texttt{data}\ \mathit{SNat}\ n\ \texttt{where} \\
\quad \mathit{Zero} :: \mathit{SNat}\ \mathit{Zero} \\
\quad \mathit{Succ} :: \mathit{SNat}\ n \to \mathit{SNat}\ (\mathit{Succ}\ n)
\end{array}
$$

$\mathit{SNat}\ n$ tiene solo un valor por cada $n$:

$$
\begin{array}{ll}
\mathit{Zero} & :: \mathit{SNat}\ \mathit{Zero} \\
\mathit{Succ}\ \mathit{Zero} & :: \mathit{SNat}\ (\mathit{Succ}\ \mathit{Zero}) \\
\mathit{Succ}\ (\mathit{Succ}\ \mathit{Zero}) & :: \mathit{SNat}\ (\mathit{Succ}\ (\mathit{Succ}\ \mathit{Zero}))
\end{array}
$$

Conociendo el largo de antemano:

$$
\begin{array}{llll}
\multicolumn{4}{l}{\mathit{fromList} :: \mathit{SNat}\ n \to [a] \to \mathit{Vec}\ a\ n} \\
\mathit{fromList}\ \mathit{Zero} & [\,] & = & \mathit{Nil} \\
\mathit{fromList}\ (\mathit{Succ}\ n) & (x : xs) & = & \mathit{Cons}\ x\ (\mathit{fromList}\ n\ xs) \\
\mathit{fromList}\ \_ & \_ & = & \mathit{error}\ \texttt{"wrong length!"}
\end{array}
$$

# Usando existenciales

```
data VecAny a where
  VecAny :: Vec a n → VecAny a

fromList :: [a] → VecAny a
fromList []       = VecAny Nil
fromList (x : xs) = case fromList xs of
                      VecAny ys → VecAny (Cons x ys)
```

# Usando existenciales

$$\mathtt{data}\ VecAny\ a\ \mathtt{where}$$
$$\quad VecAny :: Vec\ a\ n \rightarrow VecAny\ a$$

$$
\begin{aligned}
&fromList :: [a] \rightarrow VecAny\ a\\
&fromList\ [\,] \qquad = VecAny\ Nil\\
&fromList\ (x : xs) = \mathtt{case}\ fromList\ xs\ \mathtt{of}\\
&\qquad\qquad\qquad\qquad VecAny\ ys \rightarrow VecAny\ (Cons\ x\ ys)
\end{aligned}
$$

También podemos combinar ambas ideas e incluir un *SNat* en el tipo:

$$\mathtt{data}\ VecAny\ a\ \mathtt{where}$$
$$\quad VecAny :: SNat\ n \rightarrow Vec\ a\ n \rightarrow VecAny\ a$$

# Reflexión de tipos

Mediante el uso de un GADT que refleje (represente) tipos:

$$
\begin{array}{l}
\mathtt{data}\ \mathit{Type}\ t\ \mathtt{where} \\
\quad \mathit{RInt} \quad :: \mathit{Type}\ \mathit{Int} \\
\quad \mathit{RChar} :: \mathit{Type}\ \mathit{Char} \\
\quad \mathit{RList} \quad :: \mathit{Type}\ a \to \mathit{Type}\ [a] \\
\quad \mathit{RPair} \quad :: \mathit{Type}\ a \to \mathit{Type}\ b \to \mathit{Type}\ (a, b)
\end{array}
$$

es posible escribir funciones genéricas (type-indexed functions) de tipo

$$f :: \mathit{Type}\ a \to \ldots a \ldots$$

que hagan recursión en la representación de los tipos.

# Ejemplo: función de compresión

Queremos comprimir valores de tipos representados en *Type*.

$$\texttt{data}\ \mathit{Bit} = 0 \mid 1$$

$$
\begin{array}{llll}
\mathit{compress} :: \mathit{Type}\ t \to t \to [\mathit{Bit}] & & & \\
\mathit{compress}\ (\mathit{RInt}) & i & = & \mathit{compressInt}\ i \\
\mathit{compress}\ (\mathit{RChar}) & c & = & \mathit{compressChar}\ c \\
\mathit{compress}\ (\mathit{RList}\ ra) & [\,] & = & 0 : [\,] \\
\mathit{compress}\ (\mathit{RList}\ ra) & (a : as) & = & 1 : \mathit{compress}\ ra\ a \\
 & & & \quad + \!\! + \ \mathit{compress}\ (\mathit{RList}\ ra)\ as \\
\mathit{compress}\ (\mathit{RPair}\ ra\ rb) & (a, b) & = & \mathit{compress}\ ra\ a + \!\! + \ \mathit{compress}\ rb\ b
\end{array}
$$

donde

$$
\begin{array}{ll}
\mathit{compressInt} & :: \mathit{Int} \to [\mathit{Bit}] \\
\mathit{compressChar} & :: \mathit{Char} \to [\mathit{Bit}]
\end{array}
$$

son compresores para valores de *Int* y *Char*.