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:

$$
\begin{array}{lll}
val :: Int \to Expr & & \text{-- constructor} \\
add :: Expr \to Expr \to Expr & & \text{-- constructor} \\
eval :: Expr \to Int & & \text{-- observador}
\end{array}
$$

Ejemplo de EDSL

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

$$
\begin{array}{lll}
val :: Int \to Expr & & \text{-- constructor} \\
add :: Expr \to Expr \to Expr & & \text{-- constructor} \\
eval :: Expr \to Int & & \text{-- observador}
\end{array}
$$

Ejemplo de un programa en el DSL:

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

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

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

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.

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

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.

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

Deep embedding (1)

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

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

Operaciones de construcción (smart constructors):

```
val :: Int → Expr
val n = Val n

add :: Expr → Expr → Expr
add e e' = Add e e'
```

Deep embedding (2)

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

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

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:

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

Tipos Algebraicos

Notación de GADTs:

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

Notación clásica:

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

Tipos Algebraicos

Notación de GADTs:

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

Notación clásica:

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

En general:

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

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:

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

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

Tipos Algebraicos (2)

Una manera simple de definir estructuras arborescentes:

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

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

Los términos de un lenguaje son estructuras arborescentes

Tipos Algebraicos - Constructores

Volviendo al tipo de las expresiones en un deep embedding:

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

Tipos Algebraicos - Constructores

Volviendo al tipo de las expresiones en un deep embedding:

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

Los constructores se introducen al definir el tipo.

Tipos Algebraicos - Constructores

Volviendo al tipo de las expresiones en un deep embedding:

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

Los constructores se introducen al definir el tipo.

También puedo definir smart constructors

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

Tipos Algebraicos - Constructores

Volviendo al tipo de las expresiones en un deep embedding:

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

Los constructores se introducen al definir el tipo.

También puedo definir smart constructors

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

```
val x | x ⩾ 0 = Val x
```

Tipos Algebraicos - Observadores

Dado el tipo:

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

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

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

Tipos Algebraicos - Observadores

Dado el tipo:

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

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

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

Los patrones satisfacen la gramática:

$$
\begin{aligned}
pat ::= \; & \_ \\
\mid \; & variable \\
\mid \; & literal \\
\mid \; & (pat_1, ..., pat_m) \\
\mid \; & pat : pat \\
\mid \; & C\ pat_1 \ ...\ pat_n \\
\mid \; & var@pat
\end{aligned}
$$

Observadores - Alto Orden

Múltiples observadores pueden compartir un patrón de recursión:

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

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

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

Observadores - Alto Orden (2)

Puedo definir funciones de alto orden para capturar ese patrón

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

Observadores - Alto Orden (2)

Puedo definir funciones de alto orden para capturar ese patrón

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

Entonces

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

Alto Orden en Shallow Embedding

Agregamos variables a nuestro lenguaje de expresiones:

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

Alto Orden en Shallow Embedding

Agregamos variables a nuestro lenguaje de expresiones:

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

Nuestro evaluador debería poder aplicar el ambiente de variables:

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

Alto Orden en Shallow Embedding

Agregamos variables a nuestro lenguaje de expresiones:

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

Nuestro evaluador debería poder aplicar el ambiente de variables:

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

Entonces el tipo $Expr$ es:

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

Alto Orden en Shallow Embedding

Agregamos variables a nuestro lenguaje de expresiones:

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

Nuestro evaluador debería poder aplicar el ambiente de variables:

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

Entonces el tipo $Expr$ es:

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

y los constructores son funciones de alto orden

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

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:

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

Definiendo instancias para cada implementación:

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

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

Type Classes

Podemos empaquetar la API del lenguaje en una type class:

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

Definiendo instancias para cada implementación:

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

instance IExpr Expr where
  val  = Val
  add  = Add
  eval = foldExpr id (+)

instance IExpr Int where
  val  n   = n
  add x y = x + y
  eval e   = e
```

Otro Ejemplo de EDSL - Regiones Geométricas

Consideremos un lenguaje que manipula regiones geométricas formado por las 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:

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

Ejemplo de un programa en el DSL:

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

Shallow embedding

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

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

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

Deep embedding

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

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

Deep embedding

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

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

y la instancia

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

  p `inRegion` (Circle r)       = magnitude p <= r
  p `inRegion` (Outside r)      = not (p `inRegion` r)
  p `inRegion` (Union r r')     = p `inRegion` r || p `inRegion` r'
  p `inRegion` (Intersect r r') = p `inRegion` r && p `inRegion` r'
```