Programación Funcional Avanzada

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

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 vistos en el curso:

- QuickCheck
- Arrays
- Finger Trees
- Streams

Otros EDSLs en Haskell:

- 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*.
  - 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 *regiones geométricas* formado por las siguentes operaciones:

$$
\begin{aligned}
\mathit{inRegion} &:: \mathit{Point} \to \mathit{Region} \to \mathit{Bool} \\
\mathit{circle} &:: \mathit{Radius} \to \mathit{Region} \\
\mathit{outside} &:: \mathit{Region} \to \mathit{Region} \\
\cup &:: \mathit{Region} \to \mathit{Region} \to \mathit{Region} \\
\cap &:: \mathit{Region} \to \mathit{Region} \to \mathit{Region}
\end{aligned}
$$

Ejemplo de EDSL

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

$$
\begin{aligned}
\mathit{inRegion} &:: \mathit{Point} \to \mathit{Region} \to \mathit{Bool} \\
\mathit{circle} &:: \mathit{Radius} \to \mathit{Region} \\
\mathit{outside} &:: \mathit{Region} \to \mathit{Region} \\
\cup &:: \mathit{Region} \to \mathit{Region} \to \mathit{Region} \\
\cap &:: \mathit{Region} \to \mathit{Region} \to \mathit{Region}
\end{aligned}
$$

Ejemplo de un programa en el DSL:

$$
\begin{aligned}
&\mathit{aro} :: \mathit{Radius} \to \mathit{Radius} \to \mathit{Region} \\
&\mathit{aro}\ \mathit{r1}\ \mathit{r2} = \mathit{outside}\ (\mathit{circle}\ \mathit{r1}) \cap \mathit{circle}\ \mathit{r2}
\end{aligned}
$$

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).

$$\textbf{type}\ \mathit{Region} = \mathit{Point} \to \mathit{Bool}$$

$$\begin{aligned}
p\ \text{`}\mathit{inRegion}\text{`}\ r &= r\ p \\
\mathit{circle}\ r &= \lambda p \to \mathit{magnitude}\ p \leqslant r \\
\mathit{outside}\ r &= \lambda p \to \neg\ (r\ p) \\
r\ \cup\ r' &= \lambda p \to r\ p \vee r'\ p \\
r\ \cap\ r' &= \lambda p \to r\ p \wedge r'\ p
\end{aligned}$$

Deep embedding (1)

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

$$
\begin{aligned}
\textbf{data}\ \mathit{Region} = {} & \mathit{Circle} \quad \mathit{Radius} \\
\mid {} & \mathit{Outside} \quad \mathit{Region} \\
\mid {} & \mathit{Union} \quad \mathit{Region}\ \mathit{Region} \\
\mid {} & \mathit{Intersect}\ \mathit{Region}\ \mathit{Region}
\end{aligned}
$$

Operaciones de construcción (smart constructors):

$$
\begin{aligned}
\mathit{circle}\ r &= \mathit{Circle}\ r \\
\mathit{outside}\ r &= \mathit{Outside}\ r \\
r \cup r' &= \mathit{Union}\ r\ r' \\
r \cap r' &= \mathit{Intersect}\ r\ r'
\end{aligned}
$$

Deep embedding (2)

La operación *inRegion* hace las veces de función de interpretación.

$$
\begin{aligned}
p \text{ `}inRegion\text{` } (Circle\ r) &= magnitude\ p \leqslant r \\
p \text{ `}inRegion\text{` } (Outside\ r) &= \neg\ (p \text{ `}inRegion\text{` } r) \\
p \text{ `}inRegion\text{` } (Union\ r\ r') &= p \text{ `}inRegion\text{` } r \vee p \text{ `}inRegion\text{` } r' \\
p \text{ `}inRegion\text{` } (Intersect\ r\ r') &= p \text{ `}inRegion\text{` } r \wedge p \text{ `}inRegion\text{` } r'
\end{aligned}
$$

Que embedding elegir?

Shallow embedding

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

Cons: Agregar nuevas formas de interpretación (por ejemplo, computar también el área de una región) puede implicar una reimplemantación completa.

Que embedding elegir?

Shallow embedding

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

Cons: Agregar nuevas formas de interpretación (por ejemplo, computar también el área de una región) puede implicar una reimplemantación completa.

Deep embedding

Pros: Es simple agregar nuevas funciones de interpretación.

Cons: Agregar nuevas construcciones al lenguaje implica modificar el tipo del AST (el tipo *Region*) y todas 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,

$$(r \cup r') \cup r'' = r \cup (r' \cup r'')$$

$$(r \cap r') \cap r'' = r \cap (r' \cap r'')$$

DSLs fuertemente tipados (1)

El siguiente EDSL es usado por PARADISE, un DSL desarrollado en Credite Suisse que genera planillas Excel.

Es un deep embedding con el siguiente AST de expresiones aritméticas (correspondientes a celdas Excel en PARADISE):

```
data Exp = LitDbl Double
         | LitStr String
         | LitBool Bool
         | Apply Func [Exp]
         | Var Id
        deriving (Eq, Show)
type Func = String
type Id = String
```

DSLs fuertemente tipados (2)

La sintaxis de las expresiones permite construir términos bien formados como este,

$$\mathit{Apply}\ \texttt{"*"}\ [\mathit{Apply}\ \texttt{"+"}\ [\mathit{LitDbl}\ 3, \mathit{LitDbl}\ 5], \mathit{LitDbl}\ 2]$$

DSLs fuertemente tipados (2)

La sintaxis de las expresiones permite construir términos bien formados como este,

$$\mathit{Apply}\ \texttt{"*"}\ [\mathit{Apply}\ \texttt{"+"}\ [\mathit{LitDbl}\ 3, \mathit{LitDbl}\ 5], \mathit{LitDbl}\ 2]$$

pero admite también la construcción de términos "mal tipados":

$$\mathit{Apply}\ \texttt{"+"}\ [\mathit{LitDbl}\ 3, \mathit{LitStr}\ \texttt{"hola"}]$$

DSLs fuertemente tipados (2)

La sintaxis de las expresiones permite construir términos bien formados como este,

$$\mathit{Apply}\ \texttt{"*"}\ [\mathit{Apply}\ \texttt{"+"}\ [\mathit{LitDbl}\ 3, \mathit{LitDbl}\ 5], \mathit{LitDbl}\ 2]$$

pero admite también la construcción de términos "mal tipados":

$$\mathit{Apply}\ \texttt{"+"}\ [\mathit{LitDbl}\ 3, \mathit{LitStr}\ \texttt{"hola"}]$$

Como forma de remediar esto PARADISE usa el truco de declarar un **tipo fantasma** (*phantom type*) asociado al tipo de las expresiones.

$$\textbf{data}\ E\ a = E\ \mathit{Exp}$$

El tipo E simplemente actúa como un wrapper de una expresión especificando en el parámetro a un tipo para la misma.

DSLs fuertemente tipados (3)

El objetivo es que sólo se puedan construir expresiones bien tipadas.

Esto se puede lograr mediante la definición de la instancia:

```
instance Num (E Double) where
  E x + E y = E (Apply "+" [x, y])
  E x - E y = E (Apply "-" [x, y])
  E x * E y = E (Apply "*" [x, y])
  fromInteger x = E (LitDbl (fromInteger x))
```

Las operaciones de la instancia cuidan que se respete el tipado.

DSLs fuertemente tipados (3)

El objetivo es que sólo se puedan construir expresiones bien tipadas.

Esto se puede lograr mediante la definición de la instancia:

```
instance Num (E Double) where
  E x + E y = E (Apply "+" [x, y])
  E x - E y = E (Apply "-" [x, y])
  E x * E y = E (Apply "*" [x, y])
  fromInteger x = E (LitDbl (fromInteger x))
```

Las operaciones de la instancia cuidan que se respete el tipado.

Ahora se puede escribir expresiones como si fueran números regulares:

$$3 + 4 * 5 :: E\ Double$$

y Haskell construye el AST:

$$Apply\ \text{"+"}\ [LitDbl\ 3.0, Apply\ \text{"*"}\ [LitDbl\ 4.0, LitDbl\ 5.0]]$$

Parsing

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*:

$$String \rightarrow a$$

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:

$$String \rightarrow a$$

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

$$String \rightarrow [a]$$

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:

$$String \rightarrow a$$

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

$$String \rightarrow [a]$$

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

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

El tipo de un parser

En resumen,

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

El tipo de un parser

En resumen,

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

Podemos abstraer el tipo *String*:

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

- 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

$$
\begin{aligned}
\mathit{pFail} &:: && && \mathit{Parser}\ s\ a \\
\mathit{pSucceed} &:: a && && \to \mathit{Parser}\ s\ a \\
\mathit{pSym} &:: \mathit{Eq}\ s \Rightarrow s && && \to \mathit{Parser}\ s\ s \\
\texttt{<|>} &:: \mathit{Parser}\ s\ a && \to \mathit{Parser}\ s\ a && \to \mathit{Parser}\ s\ a \\
\texttt{<*>} &:: \mathit{Parser}\ s\ (a \to b) && \to \mathit{Parser}\ s\ a && \to \mathit{Parser}\ s\ b
\end{aligned}
$$

Combinadores básicos

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

Combinadores básicos

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

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

Combinadores básicos

$$
\begin{aligned}
&pFail :: Parser\ s\ a \\
&pFail = \lambda cs \rightarrow [\,]
\end{aligned}
$$

$$
\begin{aligned}
&pSucceed :: a \rightarrow Parser\ s\ a \\
&pSucceed\ a = \lambda cs \rightarrow [(a, cs)]
\end{aligned}
$$

$$
\begin{aligned}
&pSym :: Eq\ s \Rightarrow s \rightarrow Parser\ s\ s \\
&pSym\ s = \lambda cs \rightarrow \mathbf{case}\ cs\ \mathbf{of} \\
&\qquad\qquad [\,] \rightarrow [\,] \\
&\qquad\qquad (c : cs') \rightarrow \mathbf{if}\ c == s \\
&\qquad\qquad\qquad\qquad \mathbf{then}\ [(c, cs')] \\
&\qquad\qquad\qquad\qquad \mathbf{else}\ [\,]
\end{aligned}
$$

Combinadores básicos

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

Combinadores básicos

$$
\begin{aligned}
&(<|>) \quad :: \mathit{Parser}\ s\ a \to \mathit{Parser}\ s\ a \to \mathit{Parser}\ s\ a \\
&p\ <|>\ q = \lambda cs \to p\ cs +\!\!+ q\ cs
\end{aligned}
$$

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

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.

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

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.

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

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

$$pListAB = pList\ pAB$$

Otros combinadores útiles

```
(<$>)  :: (a → b) → Parser s a → Parser s b
f <$> p = pSucceed f <*> p
```

```
opt       :: Parser s a → a → Parser s a
p `opt` a = p <|> pSucceed a
```

```
pSat   :: (s → Bool) → Parser s s
pSat p = λcs → case cs of
                 []       → []
                 (c : cs') → if p c
                             then [(c, cs')]
                             else []
```

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 <$>:

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

- Definición de *pSym* usando *pSat*:

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

- Reconocer un dígito:

$$pDigit = pSat\ isDigit$$
$$\textbf{where}$$
$$isDigit\ c = (c \geqslant \texttt{'0'}) \wedge (c \leqslant \texttt{'9'})$$

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:

$$pDigit = pSat\ isDigit$$
$$\textbf{where}$$
$$isDigit\ c = (c \geqslant \texttt{'0'}) \wedge (c \leqslant \texttt{'9'})$$

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

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

Selección de resultados de parsers

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

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

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

Selección de resultados de parsers

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

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

$$(<\$) \quad :: a \to \mathit{Parser}\ s\ b \to \mathit{Parser}\ s\ a$$
$$a <\$ q = \mathit{pSucceed}\ a <* q$$

Ejemplo. Reconocer algo entre paréntesis. Dos alternativas.

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

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. Dos alternativas.

$$pParens\ p = pSym\ '('\ *> p <*\ pSym\ ')'$$

$$pParens\ p = id <\$\ pSym\ '('\ <*> p <*\ pSym\ ')'$$