Lenguajes de Dominio Específico

1/12/2025

¿Qué es?

- Un lenguaje de dominio específico (DSL: Domain Specific Language), es un lenguaje hecho para resolver un problema en un dominio particular.
- Al adaptarse a un dominio específico su uso resulta más simple (sintaxis limitada) que el de un **lenguaje de propósito general**.
- Capturan la semántica necesaria para el dominio de aplicación, también se los llama "lenguajes pequeños".
- Ejemplos: SQL, HTML, Latex, MatLab, Verilog

DSLs

Desventajas de crear un lenguaje desde cero:

- Debemos crear un parser, compilador, type-checker.
- Debemos crear herramientas para el lenguaje (syntax highlighting, testing, etc)
- Algunas construcciones están en todos los lenguajes (variables, booleanos, números, operaciones, etc).
- Definir todo esto requiere de mucho trabajo.

EDSLs

- Los DSLs embebidos (EDSLs) se integran en un lenguaje anfitrión.
- La infraestructura del lenguaje anfitrión se hereda (compilador, type-checker, etc).
- Las herramientas también.
- También hay desventajas:
  - los programas deben escribirse con la sintaxis del lenguaje anfitrión.
  - las capacidades del lenguaje anfitrión pueden limitar el diseño del edsl.

EDSLs en Haskell

Haskell es considerado un buen lenguaje anfitrión por tener:

- tipos de datos algebraicos
- funciones de alto orden
- sobrecarga de operadores
- sistema de tipo expresivo
- evaluación lazy

Hay una gran cantindad de EDSLs en Haskell (ver Hackage).

Algunos de los dominios son: generadores de parsers, pretty-printers, gráficos, animaciones, composición de música, simulaciones, diseño de hardware, concurrencia, análisis de regiones geométricas, testing, etc.

Sintaxis de un EDSL

Un EDSL en Haskell puede pensarse como tipo abstracto de datos (sintaxis) y un compilador o intérprete (semántica).

Se proveen formas para:

- **construir** elementos del tipo
- **componer** los elementos
- y **observar** elementos o **ejecutarlos**

Ejemplos

1. Librería de parsers:
   - **Tipo de datos**: `Parser`
   - **Constructores:** `item`, `failure`, `return`
   - **Combinadores:** `<|>`, `many`
   - **Ejecutar parsers:** `parse`
2. Librería de pretty printing:
   - **Tipo de datos:** `Doc`
   - **Constructores:** `empty`, `text`
   - **Combinadores:** `<>`, `sep`
   - **Ejecutar documentos:** `render`

Semántica de un EDSL

Hay dos maneras principales:

1. Deep embedding:
   - Los elementos se representan por cómo se construyen.
   - Los constructores y combinadores son triviales.
   - La estructura del EDSL está expuesta.
   - La función de ejecución (u observación) realiza el trabajo.

2. Shallow embedding:
   - Los elementos se representan por su semántica.
   - La evaluación es realizada por los constructores y combinadores.
   - La estructura del EDSL no está expuesta.
   - La función de ejecución es trivial.

La mayoría de los EDSL utilizan algo intermedio (pero cercano a un extremo).

Ejemplo 1

Interfaz de un lenguaje de expresiones aritméticas:

```
-- Tipo de datos
type Expr = ..
-- Constructores
lit :: Int -> Expr
-- Combinadores
add :: Expr -> Expr -> Expr
mult :: Expr -> Expr -> Expr
div  :: Expr -> Expr -> Expr
-- Funci'on de ejecuci'on
eval :: Expr -> Maybe Int
```

Semántica con shallow embedding

```
newtype Expr = E (Maybe Int)

lit n :: Int -> Expr
lit = E (Just n)

add :: Expr -> Expr -> Expr
add (E (Just n)) (E (Just m)) =
    E (Just (n + m))
add  _     _                  =  E Nothing

div :: Expr -> Expr -> Expr
div (E (Just n)) (E (Just m)) | m /= 0 =
    E (Just (n 'div' m))
div  _ _ = E Nothing

eval :: Expr -> Maybe Int
eval (E v) = v
```

Semántica con deep embedding

```
data Expr =  Lit Int | Add Expr Expr |
             Div Expr Expr

lit :: Int -> Expr
lit = Lit n

plus n m :: Expr -> Expr -> Expr
plus = Plus n m

div :: Expr -> Expr -> Expr
div n m = Div n m
```

Semántica con deep embedding

```
eval :: Expr -> Maybe Interfaz
eval (Lit n) = Just n
eval (Add e e') = do  n <- eval e
                      n' <- eval e'
                      return (n + n')

eval (Div e e') = do n <- eval e
                     n' <- eval e'
                     if n' ==0
                     then throw
                     else return (div n n↩
                        ')
```

Ejemplo 2

Interfaz de un EDSL para conjuntos finitos de enteros:

```
-- Tipo de datos
type IntSet = ..
-- Constructores
empty :: IntSet
insert :: Int -> IntSet -> IntSet
-- Combinadores
delete :: Int -> IntSet -> IntSet
-- Funci'on de observaci'on
member :: Int -> IntSet -> Bool
```

Semántica con deep embedding

```
data IntSet = E | INS x IntSet |
              DEL x IntSet

empty = E
insert x s = INS x s

delete x s = DEL x s

member x E = false
member x (INS n s) = x == n || member x s
member x (DEL n s) = x /= n && member x s
```

Ejemplo2: Semántica con shallow embedding

```
newtype IntSet = IS (Int -> Bool)

empty = IS (\_ -> False)
insert x (IS f) = IS (\n -> n == x || f n)

delete x (IS f) = IS (\n -> n /= x && f n)

member x (IS f) = f x
```

Ejemplo2: Semántica intermedia

```
type IntSet = [Integer]

empty = []
insert x xs = x:xs

delete x xs = filter (/= x) xs

member x xs = any (== x) xs
```

Abstracción

Para abstraer la implementación, creamos una interfaz con un módulo:

```
module IntSet
(IntSet, empty , insert, delete, member) ↩
   where
...
```

Bibliografía

- J. Gibbons. “Functional Programming for Domain-Specific Languages”. In: Central European Functional Programming School. CEFP 2013. Lecture Notes in Computer Science(), vol 8606. Springer, Cham. https://doi.org/10.1007/978-3-319-15940-9