

# Lenguajes de Dominio Específico (o Lenguajes para Todos)

Mauro Jaskelioff  
17/11/2017



**“Todo sistema que acepte entrada de un usuario es un procesador de lenguaje.”**

*John Reynolds*

- ▶ ¡Los lenguajes están en todas partes!
- ▶ En forma más o menos evidente, todo programa es un intérprete/compilador.
- ▶ Sin embargo, este no es el punto de vista que *usualmente* se tiene en un desarrollo.
- ▶ ¿Qué pasa cuando tomamos ese punto de vista?

# Lenguajes de Dominio Específico

- ▶ Dada una clase de problemas se puede diseñar un lenguaje para expresar sus soluciones.
- ▶ Un **lenguaje de dominio específico** (DSL) es un lenguaje de programación diseñado para resolver problemas de un dominio en particular.
  - ▶ A diferencia de los **lenguajes generales**, como C, Java, Haskell, etc.
- ▶ Algunos ejemplos de DSLs son: SQL, R, HTML,  $\text{\LaTeX}$ .

# Evaluando DSLs

## Ventajas:

- ▶ Programas más pequeños, claros y intuitivos.
  - ▶ Accesible a no expertos.
- ▶ Nivel de abstracción mayor.
  - ▶ Más fácil de mantener.
- ▶ Semántica restringida.
  - ▶ Más fácil de optimizar y verificar.

## Desventajas:

- ▶ Hay que hacer un parser, type-checker, compilador.
- ▶ Debido a esto, a menudo están ligados a una sola plataforma.
- ▶ Hay que escribir herramientas específicas (syntax highlighting, testing, documentación, etc.).
- ▶ Es tedioso redefinir booleanos, enteros, etc. (y sus operaciones)

# DSLs embebidos

- ▶ Un **DSL embebido** aprovecha la infraestructura de un **lenguaje anfitrión** pre-existente.
- ▶ Reusa, su sintaxis, sistema de tipos, compilador, bibliotecas, herramientas.
- ▶ Permite combinar varios EDSLs.
- ▶ Pero la complejidad del lenguaje anfitrión queda expuesta.
- ▶ El diseño queda limitado por las capacidades del lenguaje anfitrión.

# Lenguajes buenos para EDSLs

¿Qué tiene que tener un buen lenguaje anfitrión?

- ▶ Permitir la sobrecarga de operaciones (números, cadenas)
- ▶ Poder construir ASTs
- ▶ Un sistemas de tipos expresivo
- ▶ Buenas herramientas
- ▶ Buenas facilidades para la generación de código.

# Haskell como Lenguaje Anfitrión

Haskell tiene varias características que lo hacen un buen anfitrión:

- ▶ Operadores definidos por el usuario, con precedencia y asociatividad.
- ▶ Funciones de alto orden
- ▶ Números sobrecargados
- ▶ Notación **do**
- ▶ Sintaxis minimal, llamados a funciones sin paréntesis
- ▶ Extensiones para manipular desde Haskell el AST de programas en Haskell
- ▶ Evaluación perezosa

# Sintaxis de un EDSL

- ▶ Un EDSL se escribe **manipulando un tipo** de datos (a veces mas de un tipo).
- ▶ Se proveen formas de:
  - ▶ **construir** elementos del tipo
  - ▶ **componer** elementos del tipo
  - ▶ **hacer observaciones** sobre elementos del tipo (o ejecutar).
- ▶ La biblioteca de parsers es un ejemplo de EDSL.
  - ▶ Manipulamos el tipo *Parser*
  - ▶ Tenemos parsers básicos (*item*, *failure*, *return*)
  - ▶ Componemos parsers ( $\langle | \rangle$ ,  $\gg=$ )
  - ▶ Ejecutamos los parsers (*parse*)



# Más sobre Sintaxis de EDSLs

- ▶ Pretty printing
  - ▶ Manipulamos el tipo *Doc* de documentos
  - ▶ Tenemos documentos básicos (*empty*, *text*)
  - ▶ Componemos documentos (*sep*, *<>*)
  - ▶ Observamos documentos (*render*)
- ▶ A menudo los tipos sobre los que trabajamos son mónadas.
  - ▶ Por lo tanto podemos usar la notación **do**.
- ▶ Las funciones de alto orden permiten definir combinadores que representan estructuras de control.
  - ▶ El ejemplo paradigmático es  $\gg=$ .

# Diseño de EDSLs

Al diseñar un EDSL, son importantes:

- ▶ **Composicionalidad** Combinar elementos para construir cosas complejas debe ser fácil y natural.
  - ▶ Si los programas quedan raros o incómodos, tal vez sea necesario rediseñar la interfaz.
- ▶ **Abstracción** El usuario no debe necesitar saber (o poder aprovechar) la implementación.
  - ▶ Cambiar la implementación no debería romper programas del usuario.

# Semántica de un EDSL

- ▶ Shallow embedding
  - ▶ Representamos los elementos del tipo por su semántica
    - ▶ Ejemplo:  $\text{Parser } a = \text{String} \rightarrow [(a, \text{String})]$
  - ▶ Los constructores y combinadores hacen el trabajo.
  - ▶ La función de ejecución es trivial.
- ▶ Deep embedding
  - ▶ Representamos los elementos por como se construyen (el tipo es básicamente un AST).
  - ▶ El trabajo lo hace la función de ejecución.
  - ▶ Los constructores y combinadores son triviales.
  - ▶ Podemos optimizar programas, manipulando el AST.
- ▶ Nota: Estos son dos puntos extremos de un amplio espectro.
  - ▶ Los EDSLs pueden ser algo intermedio.

# Ejemplo simple

- ▶ Definimos un pequeño lenguaje para expresiones aritméticas alrededor del tipo *Expr*
- ▶ La interfaz es:

```
-- El tipo principal
type Expr

-- Construimos expresiones
lit      :: Int → Expr

-- Manipulamos expresiones
plus     :: Expr → Expr → Expr
divide   :: Expr → Expr → Expr

-- ejecutamos expresiones
runExpr :: Expr → Maybe Int
```

# Funciones primitivas y derivadas

Distinguimos entre:

- ▶ **Funciones Primitivas:** Tienen acceso a la representación interna.
- ▶ **Funciones Derivadas:** Sin acceso a la representación interna. Se definen en base a otras operaciones.

$$\begin{aligned} \textit{sucesor} &:: \textit{Expr} \rightarrow \textit{Expr} \\ \textit{sucesor } n &= \textit{plus } n \textit{ (lit 1)} \end{aligned}$$

- ▶ Es conveniente tener una cantidad mínima de funciones primitivas.

# Shallow Embedding para expresiones

**newtype** *Expr* = *E* (*Maybe Int*)

*lit* :: *Int* → *Expr*

*lit n* = *E* (*Just n*)

*plus* :: *Expr* → *Expr* → *Expr*

*plus* (*E* (*Just n*)) (*E* (*Just m*)) = *E* (*Just* (*n + m*))

*plus* \_ \_ = *E Nothing*

*divide* :: *Expr* → *Expr* → *Expr*

*divide* (*E* (*Just n*)) (*E* (*Just m*)) | *m* ≠ 0  
= *E* (*Just* (*n 'div' m*))

*divide* \_ \_ = *E Nothing*

*runExpr* :: *Expr* → *Maybe Int*

*runExpr* (*E v*) = *v*

# Deep Embedding para expresiones

**data**  $Expr = Lit\ Int \mid Plus\ Expr\ Expr \mid Divide\ Expr\ Expr$

$lit \quad \quad \quad :: Int \rightarrow Expr$

$lit\ n \quad \quad = Lit\ n$

$plus \quad \quad \quad :: Expr \rightarrow Expr \rightarrow Expr$

$plus\ n\ m \quad = Plus\ n\ m$

$divide \quad \quad :: Expr \rightarrow Expr \rightarrow Expr$

$divide\ n\ m = Divide\ n\ m$

- Los constructores y combinadores simplemente construyen el AST del lenguaje.

# Deep Embedding para expresiones (cont.)

- ▶ Todo el trabajo se hace en la función de ejecución.

```
runExpr :: Expr → Maybe Int  
runExpr (Lit i) = return i  
runExpr (Plus n m) = do vn ← runExpr n  
                        vm ← runExpr m  
                        return (vn + vm)  
runExpr (Divide n m) = do vn ← runExpr n  
                        vm ← runExpr m  
                        if vm ≠ 0  
                        then return (vn 'div' vm)  
                        else throw
```

- ▶ Desde el punto de vista del usuario, no hay diferencia entre las dos implementaciones.



# Semántica de un EDSL

- ▶ Shallow embedding
  - ▶ Representamos los elementos del tipo por su semántica
    - ▶ Ejemplo:  $\text{Parser } a = \text{String} \rightarrow [(a, \text{String})]$
  - ▶ Los constructores y combinadores hacen el trabajo.
  - ▶ La función de ejecución es trivial.
- ▶ Deep embedding
  - ▶ Representamos los elementos por como se construyen (el tipo es básicamente un AST).
  - ▶ El trabajo lo hace la función de ejecución.
  - ▶ Los constructores y combinadores son triviales.
  - ▶ Podemos optimizar programas, manipulando el AST.
- ▶ Nota: Estos son dos puntos extremos de un amplio espectro.
  - ▶ Los EDSLs pueden ser algo intermedio (usualmente cercano a un extremo).

# Shallow vs. Deep

- ▶ Shallow

- ▶ Trabajar directamente sobre la semántica es usualmente conciso y elegante.
- ▶ Se pueden usar fácilmente las características del lenguaje anfitrión, (como la recursión o el sharing).
- ▶ Más difícil de depurar y analizar.

- ▶ Deep

- ▶ Control total sobre el AST
- ▶ Diferentes interpretaciones sobre el mismo programa
  - ▶ Más fácil de depurar y analizar
- ▶ Posibilita optimizaciones mediante la transformación del AST.
- ▶ Más difícil de usar las características del lenguaje anfitrión.

# Un lenguaje para formas

**data** *Shape*

-- Funciones constructoras

*empty* :: *Shape*

*circle* :: *Shape* -- círculo unidad

*square* :: *Shape* -- cuadrado unidad

-- Combinadores

*translate* :: *Vec* → *Shape* → *Shape*

*scale* :: *Vec* → *Shape* → *Shape*

*rotate* :: *Angle* → *Shape* → *Shape*

*union* :: *Shape* → *Shape* → *Shape*

*intersect* :: *Shape* → *Shape* → *Shape*

*difference* :: *Shape* → *Shape* → *Shape*

-- Función de ejecución

*inside* :: *Point* → *Shape* → *Bool*

# Operaciones adicionales

- ▶ ¿Cuáles son primitivas y cuáles derivadas?
- ▶ Podemos extender la interfaz con otras operaciones

*invert* :: *Shape* → *Shape*

*transform* :: *Matrix* → *Shape* → *Shape*

- ▶ Y definir algunas operaciones como derivadas.

*scale* :: *Vec* → *Shape* → *Shape*

*scale v* = *transform* (*matrix* (*vecX v*) 0 0 (*vecY v*))

*rotate* :: *Angle* → *Shape* → *Shape*

*rotate a* = *transform* (*matrix* (*cos a*) (*-sin a*)  
(*sin a*) (*cos a*))

*difference* :: *Shape* → *Shape* → *Shape*

*difference a b* = *a* 'intersect' *invert b*

# Shapes: Shallow embedding

- ▶ ¿Cuáles son las observaciones que podemos hacer?

*inside* :: *Point*  $\rightarrow$  *Shape*  $\rightarrow$  *Bool*

- ▶ Proponemos entonces

**newtype** *Shape* = *Shape* (*Point*  $\rightarrow$  *Bool*)

*inside* :: *Point*  $\rightarrow$  *Shape*  $\rightarrow$  *Bool*

*inside* *p* (*Shape* *f*) = *f* *p*

- ▶ ¡No siempre será tan fácil!
  - ▶ Puede ser complicado encontrar una representación que de lugar a una semántica **composicional**.

## Shapes: Shallow embedding (cont.)

$$\begin{aligned} \text{empty} &= \text{Shape } (\lambda p \rightarrow \text{False}) \\ \text{circle} &= \text{Shape } (\lambda p \rightarrow (pt_X p)^2 + (pt_Y p)^2 \leq 1) \\ \text{square} &= \text{Shape } (\lambda p \rightarrow \text{abs } (pt_X p) \leq 1 \wedge \\ &\quad \text{abs } (pt_Y p) \leq 1) \\ \text{transform } m \ s &= \text{Shape } (\lambda p \rightarrow \text{mul } (\text{inv } m) \ p \text{ 'inside' } s) \\ \text{translate } v \ s &= \text{Shape } (\lambda p \rightarrow \text{sub } p \ v \text{ 'inside' } s) \\ \text{union } s \ t &= \text{Shape } (\lambda p \rightarrow p \text{ 'inside' } s \vee p \text{ 'inside' } t) \\ \text{intersect } s \ t &= \text{Shape } (\lambda p \rightarrow p \text{ 'inside' } s \wedge p \text{ 'inside' } t) \\ \text{invert } s &= \text{Shape } (\lambda p \rightarrow \neg (p \text{ 'inside' } s)) \end{aligned}$$

# Shapes: Deep embedding

```
data Shape = Empty | Circle | Square  
           | Translate Vec Shape  
           | Transform Matrix Shape  
           | Union Shape Shape  
           | Intersect Shape Shape  
           | Invert Shape
```

```
empty      = Empty  
circle     = Circle  
translate  = Translate  
transform  = Transform  
union      = Union  
intersect  = Intersect  
invert     = Invert
```

# Shapes: Deep embedding (cont.)

- ▶ La representación (*Shape*) es fácil.
- ▶ El trabajo está en la función de observación.

<i>inside</i>	$:: Point \rightarrow Shape \rightarrow Bool$
<i>p 'inside' Empty</i>	$= False$
<i>p 'inside' Circle</i>	$= (pt_x\ p)^2 + (pt_y\ p)^2 \leq 1$
<i>p 'inside' Square</i>	$= abs\ (pt_x\ p) \leq 1 \wedge$ $abs\ (pt_y\ p) \leq 1$
<i>p 'inside' Translate v a</i>	$= sub\ p\ v\ 'inside'\ a$
<i>p 'inside' Transform m a</i>	$= mul\ (inv\ m)\ p\ 'inside'\ a$
<i>p 'inside' Union a b</i>	$= p\ 'inside'\ a \vee p\ 'inside'\ b$
<i>p 'inside' Intersect a b</i>	$= p\ 'inside'\ a \wedge p\ 'inside'\ b$
<i>p 'inside' Invert s</i>	$= \neg (p\ 'inside'\ s)$



# Abstracción

- Ponemos el código en un módulo para poder abstraer (ocultar) los detalles internos

**module** *Shape*

  (**module** *Matrix*

    , *Shape*

    , *empty*, *circle*, *square*

    , *translate*, *transform*, *scale*, *rotate*

    , *union*, *intersect*, *difference*, *invert*

    , *inside*

  ) **where**

**import** *Matrix*

...

- La interfaz es la misma independientemente de si es un deep o un shallow embedding.

# Resumen

- ▶ Se puede pensar cualquier programa desde el punto de vista de lenguajes.
- ▶ Los DSL son un buen enfoque para el desarrollo de software.
- ▶ Embeber un lenguaje en un lenguaje anfitrión facilita mucho el trabajo.
- ▶ Haskell es un buen lenguaje para EDSLs.
- ▶ Shallow vs deep embedding.

# Referencias

- ▶ *Modular Domain Specific Languages and Tools*. Paul Hudak. Fifth International Conference on Software Reuse, pages 134142. IEEE Computer Society Press, 1998.
- ▶ *Domain-specific languages and code synthesis using Haskell*. Andy Gyll. Commun. ACM, vol. 57, no. 6, pp. 4249, June 2014, also appeared in ACM Queue, Vol 12(4), April 2014.