Lenguajes de Dominio Específico (o Lenguajes para Todos)

Mauro Jaskelioff
17/11/2017

Lenguajes para Todos

"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, 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: $\mathit{Parser}\ a = \mathit{String} \to [(a, \mathit{String})]$
  - 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 accesso a la representación interna. Se definen en base a otras operaciones.

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

- Es conveniente tener una cantidad mínima de funciones primitivas.

Shallow Embedding para expresiones

$$
\begin{aligned}
&\textbf{newtype } \mathit{Expr} = E\ (\mathit{Maybe\ Int}) \\
&\mathit{lit} && :: \mathit{Int} \to \mathit{Expr} \\
&\mathit{lit}\ n && = E\ (\mathit{Just}\ n) \\
&\mathit{plus} && :: \mathit{Expr} \to \mathit{Expr} \to \mathit{Expr} \\
&\mathit{plus}\ (E\ (\mathit{Just}\ n))\ (E\ (\mathit{Just}\ m)) && = E\ (\mathit{Just}\ (n + m)) \\
&\mathit{plus}\ \_\ \_ && = E\ \mathit{Nothing} \\
&\mathit{divide} && :: \mathit{Expr} \to \mathit{Expr} \to \mathit{Expr} \\
&\mathit{divide}\ (E\ (\mathit{Just}\ n))\ (E\ (\mathit{Just}\ m)) \mid m \neq 0 \\
&\qquad && = E\ (\mathit{Just}\ (n\ \text{`}\mathit{div}\text{`}\ m)) \\
&\mathit{divide}\ \_\ \_ && = E\ \mathit{Nothing} \\
&\mathit{runExpr} && :: \mathit{Expr} \to \mathit{Maybe\ Int} \\
&\mathit{runExpr}\ (E\ v) && = v
\end{aligned}
$$

Deep Embedding para expresiones

$$
\begin{aligned}
&\textbf{data}\ \mathit{Expr} = \mathit{Lit}\ \mathit{Int} \mid \mathit{Plus}\ \mathit{Expr}\ \mathit{Expr} \mid \mathit{Divide}\ \mathit{Expr}\ \mathit{Expr} \\
&\mathit{lit} \quad :: \mathit{Int} \to \mathit{Expr} \\
&\mathit{lit}\ n \quad = \mathit{Lit}\ n \\
&\mathit{plus} \quad :: \mathit{Expr} \to \mathit{Expr} \to \mathit{Expr} \\
&\mathit{plus}\ n\ m \quad = \mathit{Plus}\ n\ m \\
&\mathit{divide} \quad :: \mathit{Expr} \to \mathit{Expr} \to \mathit{Expr} \\
&\mathit{divide}\ n\ m = \mathit{Divide}\ n\ m
\end{aligned}
$$

- 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.

$$
\begin{aligned}
&\mathit{runExpr} && :: \mathit{Expr} \to \mathit{Maybe\ Int}\\
&\mathit{runExpr}\ (\mathit{Lit}\ i) && = \mathit{return}\ i\\
&\mathit{runExpr}\ (\mathit{Plus}\ n\ m) && = \mathbf{do}\ \mathit{vn} \leftarrow \mathit{runExpr}\ n\\
&&&\qquad \mathit{vm} \leftarrow \mathit{runExpr}\ m\\
&&&\qquad \mathit{return}\ (\mathit{vn} + \mathit{vm})\\
&\mathit{runExpr}\ (\mathit{Divide}\ n\ m) && = \mathbf{do}\ \mathit{vn} \leftarrow \mathit{runExpr}\ n\\
&&&\qquad \mathit{vm} \leftarrow \mathit{runExpr}\ m\\
&&&\qquad \mathbf{if}\ \mathit{vm} \neq 0\\
&&&\qquad\quad \mathbf{then}\ \mathit{return}\ (\mathit{vn}\ `\mathit{div}`\ \mathit{vm})\\
&&&\qquad\quad \mathbf{else}\ \mathit{throw}
\end{aligned}
$$

- 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: $Parser\ a = String \rightarrow [(a, String)]$
  - 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 carácterísticas 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

$$
\begin{aligned}
&\mathit{invert} &&:: \mathit{Shape} \to \mathit{Shape}\\
&\mathit{transform} &&:: \mathit{Matrix} \to \mathit{Shape} \to \mathit{Shape}
\end{aligned}
$$

- Y definir algunas operaciones como derivadas.

$$
\begin{aligned}
&\mathit{scale} && :: \mathit{Vec} \to \mathit{Shape} \to \mathit{Shape}\\
&\mathit{scale}\ v && = \mathit{transform}\ (\mathit{matrix}\ (\mathit{vecX}\ v)\ 0\ 0\ (\mathit{vecY}\ v))\\
&\mathit{rotate} && :: \ \mathit{Angle} \to \mathit{Shape} \to \mathit{Shape}\\
&\mathit{rotate}\ a && = \mathit{transform}\ (\mathit{matrix}\ (\mathit{cos}\ a)\ (-\mathit{sin}\ a)\\
&&& \qquad\qquad\qquad\qquad\quad (\mathit{sin}\ a)\ \ (\mathit{cos}\ a))\\
&\mathit{difference} && :: \ \mathit{Shape} \to \mathit{Shape} \to \mathit{Shape}\\
&\mathit{difference}\ a\ b = a\ `\mathit{intersect}`\ \mathit{invert}\ b
\end{aligned}
$$

Shapes: Shallow embedding

- ¿Cuáles son las observaciones que podemos hacer?

$$inside :: Point \to Shape \to Bool$$

- Proponemos entonces

$$\begin{aligned} &\textbf{newtype } Shape = Shape\ (Point \to Bool) \\ &inside :: Point \to Shape \to Bool \\ &inside\ p\ (Shape\ f) = f\ p \end{aligned}$$

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

Shapes: Shallow embedding (cont.)

$$
\begin{aligned}
\mathit{empty} &= \mathit{Shape}\ (\lambda p \to \mathit{False}) \\
\mathit{circle} &= \mathit{Shape}\ (\lambda p \to (\mathit{pt}_X\ p)^2 + (\mathit{pt}_Y\ p)^2 \leqslant 1) \\
\mathit{square} &= \mathit{Shape}\ (\lambda p \to \mathit{abs}\ (\mathit{pt}_X\ p) \leqslant 1 \wedge \\
&\qquad\qquad\qquad\qquad \mathit{abs}\ (\mathit{pt}_Y\ p) \leqslant 1) \\
\mathit{transform}\ m\ s &= \mathit{Shape}\ (\lambda p \to \mathit{mul}\ (\mathit{inv}\ m)\ p\ `\mathit{inside}`\ s) \\
\mathit{translate}\ v\ s &= \mathit{Shape}\ (\lambda p \to \mathit{sub}\ p\ v\ `\mathit{inside}`\ s) \\
\mathit{union}\ s\ t &= \mathit{Shape}\ (\lambda p \to p\ `\mathit{inside}`\ s \vee p\ `\mathit{inside}`\ t) \\
\mathit{intersect}\ s\ t &= \mathit{Shape}\ (\lambda p \to p\ `\mathit{inside}`\ s \wedge p\ `\mathit{inside}`\ t) \\
\mathit{invert}\ s &= \mathit{Shape}\ (\lambda p \to \neg\ (p\ `\mathit{inside}`\ s))
\end{aligned}
$$

Shapes: Deep embedding

$$
\begin{aligned}
\textbf{data}\ \mathit{Shape} = {} & \mathit{Empty} \mid \mathit{Circle} \mid \mathit{Square} \\
& \mid \mathit{Translate}\ \mathit{Vec}\ \mathit{Shape} \\
& \mid \mathit{Transform}\ \mathit{Matrix}\ \mathit{Shape} \\
& \mid \mathit{Union}\ \mathit{Shape}\ \mathit{Shape} \\
& \mid \mathit{Intersect}\ \mathit{Shape}\ \mathit{Shape} \\
& \mid \mathit{Invert}\ \mathit{Shape}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{empty} &= \mathit{Empty} \\
\mathit{circle} &= \mathit{Circle} \\
\mathit{translate} &= \mathit{Translate} \\
\mathit{transform} &= \mathit{Transform} \\
\mathit{union} &= \mathit{Union} \\
\mathit{intersect} &= \mathit{Intersect} \\
\mathit{invert} &= \mathit{Invert}
\end{aligned}
$$

Shapes: Deep embedding (cont.)

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

$$
\begin{array}{ll}
\mathit{inside} & :: \mathit{Point} \to \mathit{Shape} \to \mathit{Bool} \\
p\ \text{`}\mathit{inside}\text{`}\ \mathit{Empty} & = \mathit{False} \\
p\ \text{`}\mathit{inside}\text{`}\ \mathit{Circle} & = (\mathit{pt}_X\ p)^2 + (\mathit{pt}_Y\ p^2) \leqslant 1 \\
p\ \text{`}\mathit{inside}\text{`}\ \mathit{Square} & = \mathit{abs}\ (\mathit{pt}_X\ p) \leqslant 1 \wedge \\
 & \quad \mathit{abs}\ (\mathit{pt}_Y\ p \leqslant 1) \\
p\ \text{`}\mathit{inside}\text{`}\ \mathit{Translate}\ v\ a & = \mathit{sub}\ p\ v\ \text{`}\mathit{inside}\text{`}\ a \\
p\ \text{`}\mathit{inside}\text{`}\ \mathit{Transform}\ m\ a & = \mathit{mul}\ (\mathit{inv}\ m)\ p\ \text{`}\mathit{inside}\text{`}\ a \\
p\ \text{`}\mathit{inside}\text{`}\ \mathit{Union}\ a\ b & = p\ \text{`}\mathit{inside}\text{`}\ a \vee p\ \text{`}\mathit{inside}\text{`}\ b \\
p\ \text{`}\mathit{inside}\text{`}\ \mathit{Intersect}\ a\ b & = p\ \text{`}\mathit{inside}\text{`}\ a \wedge p\ \text{`}\mathit{inside}\text{`}\ b \\
p\ \text{`}\mathit{inside}\text{`}\ \mathit{Invert}\ s & = \neg\ (p\ \text{`}\mathit{inside}\text{`}\ s)
\end{array}
$$

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.