UNIVERSITY OF
OXFORD

Embedded Domain-Specific Languages

Jeremy Gibbons
FLOLAC, Taipei, July 2014

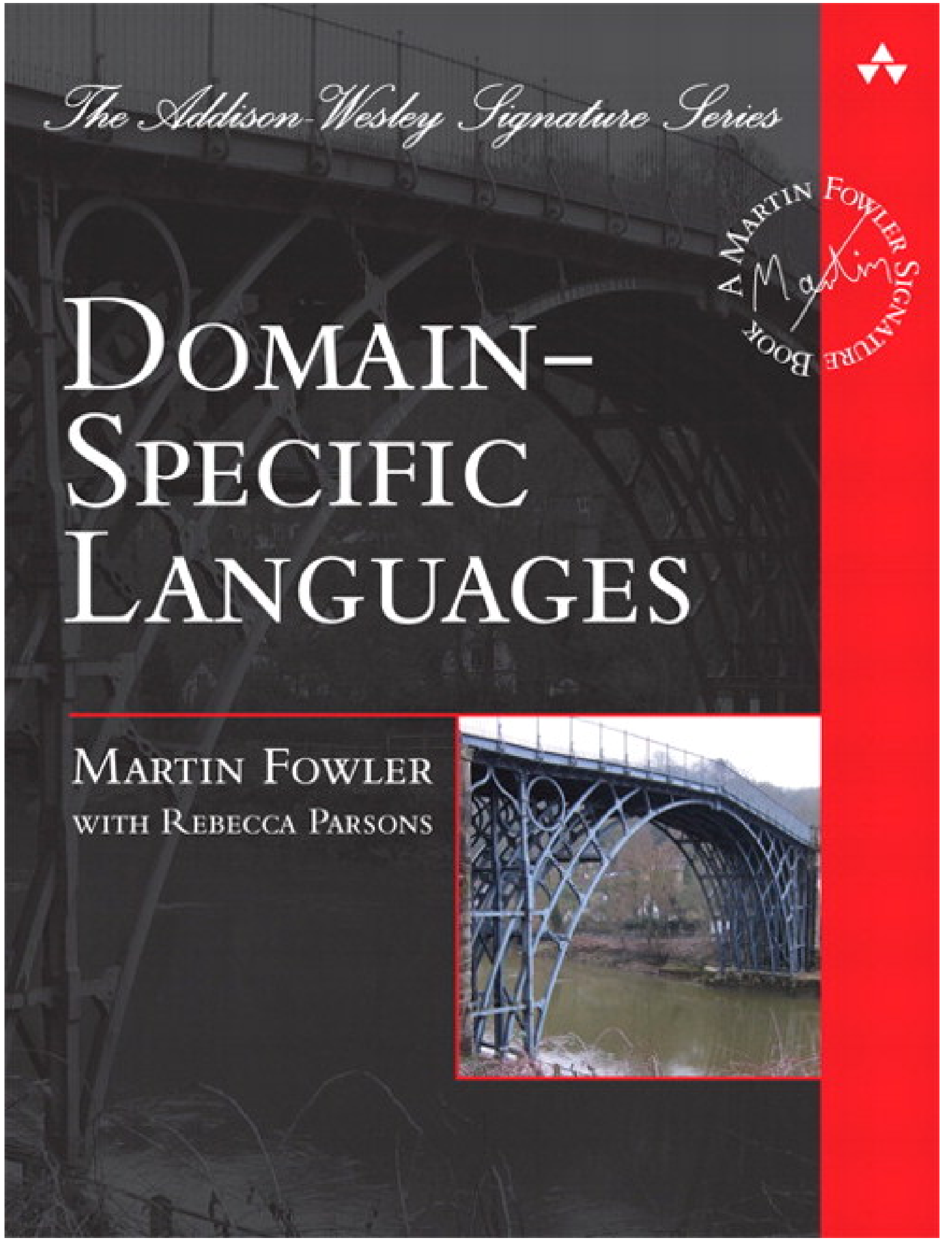
宜學藝

— 農民曆，今天

1. Introduction

“***Domain-specific language***: a computer programming language of limited expressiveness focussed on a particular domain” (Fowler)

- customized for domain
- common assumptions wired in
- more direct, less general



1.1. History

1980s: “fourth-generation languages”

1970s: Bentley’s “little languages” in Unix

1960s: “application-oriented”, “task-specific”, “special purpose”

1950s: Fortran, Cobol. . . ?

Not a new idea!

1.2. Approaches

Standalone:

- \+ custom syntax; no favoured implementation language
- \+ standard compilation techniques
- \+ may be diagrammatic, gestural...
- – significant effort, reinvented wheels

Embedded (our focus):

- \+ reuse features of host language
- \+ familiar notation
- – awkward notation
- – still “programming”
- – leaky abstractions

1.3. Embedding approaches

Deep embedding:

- terms construct ASTs
- operational
- syntax-driven

Shallow embedding:

- terms are directly interpreted
- denotational
- semantics-driven

1.4. FP support for embedded DSLs

Most work in OO on DSLs assumes standalone approach.
Much work in FP assumes embedded.

Why is that?

- *algebraic datatypes:* lightweight definitions of tree-shaped data
- *higher-order functions:* programs parametrized by other programs

2. Algebraic datatypes for DSLs

Deep embedding centred around ASTs.

Lightweight algebraic datatypes an essential feature:

- observers inductively defined over structure
- optimizations and transformations via tree manipulation

(Incidentally, algebraic datatypes also very convenient as a marshalling format for interoperation.)

2.1. A simple language

A *deeply embedded* expression language:

```
data ExprD :: ∗ where
  Val  :: Integer           → ExprD
  Add  :: ExprD → ExprD → ExprD
```

For example, the expression $3 + (4 + 5)$ is represented by the term *Add* (*Val* 3) (*Add* (*Val* 4) (*Val* 5)), which has this shape:

```
    +
   / \
  3   +
     / \
    4   5
```

2.2. One semantics

To evaluate an *ExprD*, yielding an *Integer*:

```
eval :: ExprD → Integer
eval (Val n)     = n
eval (Add x y) = eval x + eval y
```

2.3. Another semantics

To print an *ExprD*, yielding a *String*:

```
print :: ExprD → String
print (Val n)     = show n
print (Add x y) = paren (print x ⧺ print y)
```

where

```
paren :: String → String
paren s = "(" ⧺ s ⧺ ")"
```

2.4. Deep embedding—summary

- syntax of language represented by *algebraic datatypes*
- semantics expressed by *recursive functions*
- easy to provide multiple semantics

3. Shallow embedding

Here's an alternative representation of expressions: as their evaluation.

$$\begin{aligned}
&\mathbf{type}\ \mathit{ExprS}_1 = \mathit{Integer}\\
&\mathit{val}\ :: \mathit{Integer} \rightarrow \mathit{ExprS}_1\\
&\mathit{val}\ n = n\\
&\mathit{add} :: \mathit{ExprS}_1 \rightarrow \mathit{ExprS}_1 \rightarrow \mathit{ExprS}_1\\
&\mathit{add}\ x\ y = x + y
\end{aligned}$$

Now the evaluation semantics is easy:

$$\begin{aligned}
&\mathit{eval} :: \mathit{ExprS}_1 \rightarrow \mathit{Integer}\\
&\mathit{eval}\ x = x \qquad \text{-- !}
\end{aligned}$$

The syntax has been discarded; *only semantics* is left.

3.1. Another shallow embedding

This time, under *print* interpretation:

```
type ExprS2 = String

val  :: Integer → ExprS2
val n = show n

add :: ExprS2 → ExprS2 → ExprS2
add x y = paren (x ++ "+" ++ y)

print :: ExprS2 → String
print x = x        -- !
```

3.2. Deep versus shallow embedding

Deep:

- syntax of language represented by algebraic datatypes
- semantics expressed by recursive functions
- easy to provide multiple interpretations

Shallow:

- no explicit representation of syntax, *only semantics*
- *no separate 'observers'* required
- but what about multiple interpretations?

4. Higher-order functions for DSLs

What about both interpretations at once?

```
type ExprS3 = (Integer, String)

eval :: ExprS3 → Integer
eval (n, s) = n

print :: ExprS3 → String
print (n, s) = s

val  :: Integer           → ExprS3
val n = (n, show n)

add :: ExprS3 → ExprS3 → ExprS3
add x y = (eval x + eval y, paren (print x ⧺ "+" ⧺ print y))
```

Note that with lazy evaluation, if only one interpretation is demanded, then only that one will be computed.

But with three interpretations? Ten? Unforeseen interpretations?

4.1. What makes an interpretation?

What do the different interpretations have in common?
More importantly, how do they differ?

- a *semantic domain*
- an *interpretation of values* in this domain (a function)
- an *interpretation of addition* in this domain (a binary operator)

So let's capture these ingredients:

$$\mathbf{type}\ \mathit{ExprAlg}\ a = (\mathit{Integer} \rightarrow a, a \rightarrow a \rightarrow a)$$

In mathematical terms, the ingredients of an interpretation are an 'algebra'.

4.2. Parametrized interpretation of shallow embedding

Now, a term is represented as a *parametrized interpretation*:
if you tell it how to interpret, it will give you back the interpretation.

```
type ExprS a = ExprAlg a → a

val  :: Integer               → ExprS a
val n = λ(f, g) → f n

add :: ExprS a → ExprS a → ExprS a
add x y = λ(f, g) → g (x (f, g)) (y (f, g))
```

For example,

```
e :: ExprS a
e = add (val 3) (add (val 4) (val 5))
```

4.3. Instantiating the parametrized interpretation

It’s quite general:

```
evalAlg :: ExprAlg Integer
evalAlg = (id, (+))

printAlg :: ExprAlg String
printAlg = (show, λs t → paren (s ⧺ "+" ⧺ t))
```

So with `e :: ExprS a` as before, we have

```
e evalAlg  = 12
e printAlg = "(3+(4+5))"
```

4.4. Church encoding

Where did *ExprAlg* come from?

Consider fold function for *Expr* algebraic datatype:

$$
\begin{aligned}
&\mathit{fold} :: (\mathit{Integer} \to a, a \to a \to a) \to \mathit{Expr} \to a\\
&\mathit{fold}\ (f, g)\ (\mathit{Val}\ n) \quad\ \ = f\ n\\
&\mathit{fold}\ (f, g)\ (\mathit{Add}\ x\ y) = g\ (\mathit{fold}\ (f, g)\ x)\ (\mathit{fold}\ (f, g)\ y)
\end{aligned}
$$

Swap the arguments around:

$$
\begin{aligned}
&\mathit{flipFold} :: \mathit{Expr} \to (\mathit{Integer} \to a, a \to a \to a) \to a\\
&\mathit{flipFold} :: \mathit{Expr} \to (\forall a. \mathit{ExprAlg}\ a \to a)\\
&\mathit{flipFold}\ (\mathit{Val}\ n) \quad\ \ (f, g) = f\ n\\
&\mathit{flipFold}\ (\mathit{Add}\ x\ y)\ (f, g) = g\ (\mathit{flipFold}\ x\ (f, g))\ (\mathit{flipFold}\ y\ (f, g))
\end{aligned}
$$

This is known as the *Church encoding* of e,
and $\forall a. \mathit{ExprAlg}\ a$ the Church encoding of datatype *Expr*.

4.5. Polymorphic interpretation of shallow embedding

Alternatively, using *type classes* (poor person's modules):

```
class Expr a where
  val  :: Integer → a
  add  :: a → a       → a
```

Interpretations at *Integer* and *String* types:

```
instance Expr Integer where
  val n     = n
  add x y   = x + y
instance Expr String where
  val n     = show n
  add x y   = paren (x ⧺ "+" ⧺ y)
```

Then DSL term has polymorphic type:

$$\begin{aligned}&expr :: Expr\ a \Rightarrow a\\&expr = add\ (val\ 3)\ (add\ (val\ 4)\ (val\ 5))\end{aligned}$$

and can be interpreted at any type in the type class *Expr*:

$$\begin{aligned}&evalExpr && :: Integer\\&evalExpr && = expr\end{aligned}$$

$$\begin{aligned}&printExpr && :: String\\&printExpr && = expr\end{aligned}$$