Under consideration for publication in J. Functional Programming

FUNCTIONAL PEARLS

Shallow EDSLs and Object-Oriented Programming: Beyond Simple Compositionality

WEIXIN ZHANG and BRUNO C. D. S. OLIVEIRA
The University of Hong Kong, Hong Kong

Abstract

Shallow embedded domain-specific languages (EDSLs) use *procedural abstraction* to directly encode a DSL into an existing host language. Procedural abstraction has been argued to be the essence of object-oriented programming (OOP). This pearl argues that OOP abstractions (including *inheritance*, *subtyping*, and *type-refinement*) increase the modularity and reuse of shallow EDSLs when compared to classical procedural abstraction. We make this argument by taking a recent paper by Gibbons and Wu, where procedural abstraction is used in Haskell to model a simple shallow EDSL, and we recode that EDSL in Scala. To further illustrate the applicability of our OOP approach, we conduct a case study on refactoring a deep external SQL DSL implementation to make it more modular, shallow, and embedded without sacrificing performance.

1 Introduction

Since Hudak's seminal paper (1998) on embedded domain-specific languages (EDSLs), existing languages have been used to directly encode DSLs. Two common approaches to EDSLs are the so-called *shallow* and *deep* embeddings. Deep embeddings emphasize a *syntax*-first approach: the abstract syntax is defined first using a data type, and then interpretations of the abstract syntax follow. The role of interpretations in deep embeddings is to map syntactic values into semantic values in a semantic domain. Shallow embeddings emphasize a *semantics*-first approach, where a semantic domain is defined first. In the shallow approach, the operations of the EDSLs are interpreted directly into the semantic domain. Therefore there is no data type representing uninterpreted abstract syntax.

The trade-offs between shallow and deep embeddings have been widely discussed (Svenningsson & Axelsson, 2012; Jovanovic *et al.*, 2014): deep embeddings enable tranformations on the abstract syntax, and multiple interpretations are easy to implement; shallow-embeddings enforce the property of *compositionality* by construction, and are easily extended with new EDSL operations. Such discussions lead to a generally accepted belief that it is hard to support multiple interpretations and transformations in shallow embeddings.

Compositionality is considered a sign of good language design, and it is one of the hallmarks of denotational semantics. Compositionality means that the denotation of a program is constructed from denotations of its parts. One advantage of compositionality is that it leads to a modular semantics, where adding new language constructs does not

require changes in the semantics of existing constructs. Because compositionality offers a guideline for good language design, some authors (Erwig & Walkingshaw, 2014) argue that a semantics-first approach to EDSLs is superior to a syntax-first approach. In such semantics-driven approach, the idea is to first find target domain that leads to a compositional denotational semantics, and later grow the syntax on top of the semantic core. Shallow embeddings fit well with such a semantics-driven approach. Nevertheless, the limitations of shallow embeddings compared to deep embeddings can deter their use.

This functional pearl shows that, given adequate language support, supporting multiple modular interpretations in shallow DSLs is not only possible, but simple. Therefore this pearl aims to debunk the belief that multiple interpretations are hard to model with shallow embeddings. Several previous authors (Gibbons & Wu, 2014; Erwig & Walkingshaw, 2014) already observed that, in conventional functional programming, by using products and projections, multiple interpretations can be supported. Nevertheless, the use of products and projections is very cumbersome, and often leads to code that is not modular. We argue that multiple interpretations can be encoded naturally when the host language supports common OO features, such as *subtyping*, *inheritance*, and *type-refinement*.

At the center of this pearl is Reynolds (1975) idea of *procedural abstraction*, which enables us to directly relate shallow embeddings and OOP. With procedural abstraction, data is characterized by the operations that are performed over it. This pearl starts by discussing two independently observed connections to procedural abstraction:

$$\text{Shallow Embeddings} \xleftarrow{\text{Gibbons and Wu (2014)}} \text{Procedural Abstraction} \xleftarrow{\text{Cook (2009)}} \text{OOP}$$

The first connection is between procedural abstraction and shallow embeddings. As Gibbons and Wu (2014) state "*it was probably known to Reynolds, who contrasted deep embeddings (user defined types) and shallow (procedural data structures)*". Gibbons and Wu noted the connection between shallow embeddings and procedural abstractions, although they did not go into a lot of detail. The second connection is the connection between OOP and procedural abstraction, which was widely discussed by Cook (2009).

To make our arguments we take the examples in Gibbons and Wu (2014)'s paper, where procedural abstraction is used in Haskell to model a simple *shallow* EDSL. We recode that EDSL in Scala using a recently proposed design pattern (Wang & Oliveira, 2016), which provides a simple solution to the *Expression Problem* (Wadler, 1998). From the *modularity* point of view the resulting Scala version has clear advantages over the Haskell version, due to the use of subtyping, inheritance, and type-refinement. In particular, the Scala code allows the denotation of a program to easily *depend on other modular denotations*.

While the technique proposed here does not deal with transformations, yielding efficient shallow EDSL is still possible via staging (Rompf & Odersky, 2010; Carette *et al.*, 2009). By removing the limitation of multiple interpretations, we enlarge the applicability of shallow embeddings. A concrete example is our case study, which refactors an external DSL that employs deep embedding techniques (Rompf & Amin, 2015) into a shallow EDSL. The refactored implementation allows both new interpretations and new constructs to be introduced modularly without sacrificing performance. Complete code for all examples and the case study is available online:

`https://github.com/wxzh/shallow-dsl`

⟨*circuit*⟩ ::= '`id`' ⟨*positive-number*⟩
| '`fan`' ⟨*positive-number*⟩
| '`beside`' ⟨*circuit*⟩ ⟨*circuit*⟩
| '`above`' ⟨*circuit*⟩ ⟨*circuit*⟩
| '`stretch`' ⟨*positive-numbers*⟩ ⟨*circuit*⟩

Fig. 1: The grammar of SCANS

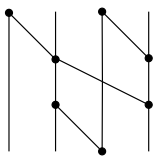

Fig. 2: The Brent-Kung circuit of width 4

2 Shallow object-oriented programming

This section shows that an OOP approach and a shallow embedding using procedural abstraction are closely related. We use the same DSL presented by Gibbons and Wu (2014) as a running example. We first give the original shallow embedded implementation in Haskell, and rewrite it towards an "OOP style". Then translating the program into an OOP language like Scala becomes straightforward.

2.1 SCANS*: A DSL for parallel prefix circuits*

SCANS (Hinze, 2004) is a DSL for describing parallel prefix circuits. Given an associative binary operator $\bullet$, the prefix sum of a non-empty sequence $x_1, x_2, ..., x_n$ is $x_1, x_1 \bullet x_2, ..., x_1 \bullet x_2 \bullet ... \bullet x_n$. Such computation can be performed in parallel for a parallel prefix circuit. Parallel prefix circuits have many applications, including binary addition and sorting algorithms. The grammar of SCANS is given in Fig. 1. SCANS has five constructs: two primitives (*id* and *fan*) and three combinators (*beside*, *above* and *stretch*). Their meanings are: *id n* contains *n* parallel wires; *fan n* has *n* parallel wires with the leftmost wire connected to all other wires from top to bottom; *beside* c_1 c_2 joins two circuits c_1 and c_2 horizontally; *above* c_1 c_2 combines two circuits of the same width vertically; *stretch ns c* inserts wires into the circuit *c* by summing up *ns*. Fig. 2 visualizes a circuit constructed using all these five constructs. The structure of this circuit is explained as follows. The whole circuit is vertically composed by three sub-circuits: the top sub-circuit is a two 2-*fan*s put side by side; the middle sub-circuit is a 2-*fan* stretched by inserting a wire on the left-hand side of its first and second wire; the bottom sub-circuit is a 2-*fan* in the middle of two 1-*id*s.

2.2 Shallow embeddings and OOP

Shallow embeddings define a language directly by encoding its semantics using procedural abstraction. In the case of SCANS, a shallow embedded implementation (in Haskell) conforms to the following types:

```
type Circuit = ...    -- the operations we wish to support for circuits
id      :: Int → Circuit
fan     :: Int → Circuit
beside  :: Circuit → Circuit → Circuit
above   :: Circuit → Circuit → Circuit
stretch :: [Int] → Circuit → Circuit
```

The type *Circuit*, representing the semantic domain, is to be filled with a concrete type according to the semantics. Each construct is declared as a function that produces a *Circuit*. Suppose that the semantics of SCANS calculates the width of a circuit. The definitions are:

$$
\begin{array}{ll}
\textbf{type}\ \mathit{Circuit} = \mathit{Int} \\
\mathit{id}\ n & = n \\
\mathit{fan}\ n & = n \\
\mathit{beside}\ c_1\ c_2 & = c_1 + c_2 \\
\mathit{above}\ c_1\ c_2 & = c_1 \\
\mathit{stretch}\ ns\ c & = \mathit{sum}\ ns
\end{array}
$$

Now we are able to construct the circuit in Fig. 2 using these definitions:

$$(\mathit{fan}\ 2\ `\mathit{beside}`\ \mathit{fan}\ 2)\ `\mathit{above}`\ \mathit{stretch}\ [2,2]\ (\mathit{fan}\ 2)\ `\mathit{above}`\ (\mathit{id}\ 1\ `\mathit{beside}`\ \mathit{fan}\ 2\ `\mathit{beside}`\ \mathit{id}\ 1)$$

For this interpretation, the Haskell domain is simply *Int*. This means that we will get the width right after the construction of a circuit (e.g. 4 for the circuit above). Note that the *Int* domain for *width* is a degenerate case of procedural abstraction: *Int* can be viewed as a no argument function. In Haskell, due to laziness, *Int* is a good representation. In a call-by-value language, a no-argument function $() \to \mathit{Int}$ is more appropriate to deal correctly with potential control-flow language constructs.

Towards OOP A simple, *semantics preserving* rewriting of *width* is given below, where a record with one field captures the domain and is declared as a **newtype**:

$$
\begin{array}{ll}
\textbf{newtype}\ \mathit{Circuit}_1 = \mathit{Circuit}_1\ \{\mathit{width}_1 :: \mathit{Int}\} \\
\mathit{id}_1\ n & = \mathit{Circuit}_1\ \{\mathit{width}_1 = n\} \\
\mathit{fan}_1\ n & = \mathit{Circuit}_1\ \{\mathit{width}_1 = n\} \\
\mathit{beside}_1\ c_1\ c_2 & = \mathit{Circuit}_1\ \{\mathit{width}_1 = \mathit{width}_1\ c_1 + \mathit{width}_1\ c_2\} \\
\mathit{above}_1\ c_1\ c_2 & = \mathit{Circuit}_1\ \{\mathit{width}_1 = \mathit{width}_1\ c_1\} \\
\mathit{stretch}_1\ ns\ c & = \mathit{Circuit}_1\ \{\mathit{width}_1 = \mathit{sum}\ ns\}
\end{array}
$$

The implementation is still shallow because Haskell's **newtype** does not add any operational behavior to the program. Hence the two programs are effectively the same. However, having fields makes the program look more like an OO program.

Porting to Scala Indeed, we can easily translate the program from Haskell to Scala:

```
trait Circuit1 {
  def width: Int
}
trait Id1 extends Circuit1 {
  val n: Int
  def width = n
}
trait Fan1 extends Circuit1 {
  val n: Int
  def width = n
}
trait Beside1 extends Circuit1 {
  val c1, c2: Circuit1
  def width = c1.width + c2.width
}
trait Above1 extends Circuit1 {
  val c1, c2: Circuit1
  def width = c1.width
}
trait Stretch1 extends Circuit1 {
  val ns: List[Int]; val c: Circuit1
  def width = ns.sum
}
```

The idea is to map Haskell's record types into an object interface (modeled as a **trait** in Scala) $Circuit_1$, and Haskell's field declarations become method declarations. Object interfaces make the connection to procedural abstraction clear: data is modeled by the operations that can be performed over it. Each case in the semantic function corresponds to a concrete implementation of $Circuit_1$, where function parameters are captured as fields.

This implementation is essentially how we would model SCANS with an OOP language in the first place. A minor difference is the use of traits instead of classes in implementing $Circuit_1$. Although a class definition like

$$\textbf{class}\ Id_1\,(n:Int)\ \textbf{extends}\ Circuit_1\,\{\textbf{def}\ width = n\}$$

is more common, some modularity offered by the trait version (e.g. multiple inheritance) is lost. To use this Scala implementation in a manner similar to the Haskell implementation, we need some smart constructors for creating objects conveniently:

$$\begin{array}{ll}
\textbf{def}\ id\,(x:Int) & = \textbf{new}\ Id_1 \quad \{\textbf{val}\ n = x\} \\
\textbf{def}\ fan\,(x:Int) & = \textbf{new}\ Fan_1 \quad \{\textbf{val}\ n = x\} \\
\textbf{def}\ above\,(x:Circuit_1, y:Circuit_1) & = \textbf{new}\ Above_1 \quad \{\textbf{val}\ c_1 = x; \textbf{val}\ c_2 = y\} \\
\textbf{def}\ beside\,(x:Circuit_1, y:Circuit_1) & = \textbf{new}\ Beside_1 \quad \{\textbf{val}\ c_1 = x; \textbf{val}\ c_2 = y\} \\
\textbf{def}\ stretch\,(x:Circuit_1, xs:Int*) & = \textbf{new}\ Stretch_1 \quad \{\textbf{val}\ ns = xs.toList; \textbf{val}\ c = x\}
\end{array}$$

Now we are able to construct the circuit shown in Fig. 2 in Scala:

$$\begin{array}{l}
\textbf{val}\ c = above\,(beside\,(fan\,(2), fan\,(2)), \\
\qquad\qquad above\,(stretch\,(fan\,(2), 2, 2), \\
\qquad\qquad\qquad beside\,(beside\,(id\,(1), fan\,(2)), id\,(1))))
\end{array}$$

Finally, calling $c.width$ will return 4 as expected.

As this example illustrates, shallow embeddings and straightforward OO programming are closely related. The syntax of the Scala code is not as concise as the Haskell version due to some extra verbosity caused by trait declarations and smart constructors. Nevertheless, the code is still quite compact and elegant, and the Scala implementation has advantages in terms of modularity, as we shall see next.

3 Interpretations in shallow embeddings

An often stated limitation of shallow embeddings is that multiple interpretations are difficult. Gibbons and Wu (2014) work around this problem by using tuples. However, their encoding needs to modify the original code, and thus is non-modular. This section illustrates how various types of interpretations can be *modularly* defined using standard OOP techniques by comparing with Gibbons and Wu's Haskell implementations.

3.1 Multiple interpretations

A single interpretation may not be enough for realistic DSLs. For example, besides *width*, we may want to have another interpretation that calculates the depth of a circuit in SCANS.

Multiple interpretations in Haskell Here is Gibbons and Wu (2014)'s solution:

$$\begin{array}{ll}
\textbf{type}\ Circuit_2 = (Int, Int) & \\
id_2\ n & = (n, 0)
\end{array}$$

$$
\begin{array}{l}
\mathit{fan}_2\ n \quad\quad\quad = (n, 1) \\
\mathit{above}_2\ c_1\ c_2\ = (\mathit{width}\ c_1, \mathit{depth}\ c_1 + \mathit{depth}\ c_2) \\
\mathit{beside}_2\ c_1\ c_2\ = (\mathit{width}\ c_1 + \mathit{width}\ c_2, \mathit{depth}\ c_1\ `\mathit{max}`\ \mathit{depth}\ c_2) \\
\mathit{stretch}_2\ \mathit{ns}\ c\ = (\mathit{sum}\ \mathit{ns}, \mathit{depth}\ c) \\
\mathit{width} = \mathit{fst} \\
\mathit{depth} = \mathit{snd}
\end{array}
$$

A tuple is used to accommodate multiple interpretations, and each interpretation is defined as a projection on the tuple. However, this solution is not modular because it relies on defining the two interpretations (*width* and *depth*) simultaneously. It is not possible to reuse the independently defined *width* interpretation in Section 2.2. Whenever a new interpretation is needed (e.g. *depth*), the original code has to be revised: the arity of the tuple must be incremented and the new interpretation has to be appended to each case.

Multiple interpretations in Scala In contrast, an OO language like Scala allows new interpretations to be introduced in a modular way:

```
trait Circuit2 extends Circuit1 {def depth:Int} // the extended semantic domain
trait Id2 extends Id1 with Circuit2 {def depth = 0}
trait Fan2 extends Fan1 with Circuit2 {def depth = 1}
trait Above2 extends Above1 with Circuit2 {
  override val c1,c2 : Circuit2
  def depth = c1.depth + c2.depth
}
trait Beside2 extends Beside1 with Circuit2 {
  override val c1,c2 : Circuit2
  def depth = Math.max (c1.depth, c2.depth)
}
trait Stretch2 extends Stretch1 with Circuit2 {
  override val c : Circuit2
  def depth = c.depth
}
```

The encoding relies on three OOP abstraction mechanisms: *inheritance*, *subtyping*, and *type-refinement*. Specifically, $\mathit{Circuit}_2$ is a subtype of $\mathit{Circuit}_1$ and declares a new method *depth*. Concrete cases, for instance Above_2, implement $\mathit{Circuit}_2$ by inheriting Above_1 and complementing the definition of *depth*. Also, fields of type $\mathit{Circuit}_1$ are covariantly refined as type $\mathit{Circuit}_2$ to allow *depth* invocations. Importantly, all definitions for *width* in Section 2.2 are *modularly reused* here.

3.2 Dependent interpretations

Dependent interpretations are a generalization of multiple interpretations. A dependent interpretation does not only depend on itself but also on other interpretations. An instance of dependent interpretation is *wellSized*, which checks whether a circuit is constructed correctly. The interpretation of *wellSized* is dependent because combinators like *above* use *width* in their definitions.

Dependent interpretations in Haskell In Haskell, dependent interpretations are again defined with tuples in a non-modular way:

$$
\begin{aligned}
&\textbf{type } \mathit{Circuit}_3 = (\mathit{Int}, \mathit{Bool})\\
&\mathit{id}_3\ n = (n, \mathit{True})\\
&\mathit{fan}_3\ n = (n, \mathit{True})\\
&\mathit{above}_3\ c_1\ c_2 = (\mathit{width}\ c_1, \mathit{wellSized}\ c_1 \wedge \mathit{wellSized}\ c_2 \wedge \mathit{width}\ c_1 \equiv \mathit{width}\ c_2)\\
&\mathit{beside}_3\ c_1\ c_2 = (\mathit{width}\ c_1 + \mathit{width}\ c_2, \mathit{wellSized}\ c_1 \wedge \mathit{wellSized}\ c_2)\\
&\mathit{stretch}_3\ ns\ c = (\mathit{sum}\ ns, \mathit{wellSized}\ c \wedge \mathit{length}\ ns \equiv \mathit{width}\ c)\\
&\mathit{wellSized} = \mathit{snd}
\end{aligned}
$$

where *width* is called in the definition of *wellSized* for above_3 and $\mathit{stretch}_3$.

Dependent interpretations in Scala Once again, it is easy to model dependent interpretation with a simple OO approach:

```
trait Circuit3 extends Circuit1 {def wellSized: Boolean} // the semantic domain
trait Id3 extends Id1 with Circuit3 {def wellSized = true}
trait Fan3 extends Fan1 with Circuit3 {def wellSized = true}
trait Above3 extends Above1 with Circuit3 {
  override val c1, c2: Circuit3
  def wellSized = c1.wellSized ∧ c2.wellSized ∧ c1.width ≡ c2.width // width dependency
}
trait Beside3 extends Beside1 with Circuit3 {
  override val c1, c2: Circuit3
  def wellSized = c1.wellSized ∧ c2.wellSized
}
trait Stretch3 extends Stretch1 with Circuit3 {
  override val c: Circuit3
  def wellSized = c.wellSized ∧ ns.length ≡ c.width // width dependency
}
```

Note that *width* and *wellSized* are defined separately. Essentially, it is sufficient to define *wellSized* while knowing only the signature of *width* in the object interface. In the definition of Above_3, for example, it is possible not only to call *wellSized*, but also *width*.

3.3 Context-sensitive interpretations

Interpretations may rely on some context. Consider an interpretation that simplifies the representation of a circuit. A circuit can be divided horizontally into layers. Each layer can be represented as a sequence of pairs (i, j), denoting the connection from wire i to wire j. For instance, the circuit shown in Fig. 2 has the following layout:

$$[[(0,1),(2,3)],[(1,3)],[(1,2)]]$$

The combinator *stretch* and *beside* will change the layout of a circuit. For example, if two circuits are put side by side, all the indices of the right circuit will be increased by the width of the left circuit. Hence the interpretation *layout* is also dependent, relying on itself as well as *width*. An intuitive implementation of *layout* performs these changes immediately to the

affected circuit. A more efficient implementation accumulates these changes and applies them all at once. Therefore, an accumulating parameter is used to achieve this goal, which makes *layout* context-sensitive.

Context-sensitive interpretations in Haskell The following Haskell code implements (non-modular) *layout*:

$$
\begin{array}{l}
\textbf{type } \mathit{Circuit}_4 = (\mathit{Int}, (\mathit{Int} \to \mathit{Int}) \to [[(\mathit{Int}, \mathit{Int})]]) \\
\mathit{id}_4\ n \qquad\qquad = (n, \lambda f \to [\,]) \\
\mathit{fan}_4\ n \qquad\qquad = (n, \lambda f \to [[(f\ 0, f\ j) \mid j \leftarrow [1\,..\,n-1]]]) \\
\mathit{above}_4\ c_1\ c_2 \;= (\mathit{width}\ c_1, \lambda f \to \mathit{layout}\ c_1\ f +\!\!+ \mathit{layout}\ c_2\ f) \\
\mathit{beside}_4\ c_1\ c_2 = (\mathit{width}\ c_1 + \mathit{width}\ c_2, \lambda f \to \mathit{lzw}\ (+\!\!+)\ (\mathit{layout}\ c_1\ f)\ (\mathit{layout}\ c_2\ (f \circ (\mathit{width}\ c_1 +)))) \\
\mathit{stretch}_4\ ns\ c \;= (\mathit{sum}\ ns, \lambda f \to \mathit{layout}\ c\ (f \circ \mathit{pred} \circ (\mathit{vs}!!))) \\
\quad \textbf{where } \mathit{vs} = \mathit{scanl1}\ (+)\ ns \\
\mathit{lzw} \qquad\qquad :: (a \to a \to a) \to [a] \to [a] \to [a] \\
\mathit{lzw}\ f\ [\,]\ ys \qquad = ys \\
\mathit{lzw}\ f\ xs\ [\,] \qquad = xs \\
\mathit{lzw}\ f\ (x : xs)\ (y : ys) = f\ x\ y : \mathit{lzw}\ f\ xs\ ys \\
\mathit{layout} = \mathit{snd}
\end{array}
$$

The domain of *layout* is a function type $(\mathit{Int} \to \mathit{Int}) \to [[(\mathit{Int}, \mathit{Int})]]$, which takes a transformation on wires and produces a layout. An anonymous function is hence defined for each case, where f is the accumulating parameter. Note that f is accumulated in beside_4 and $\mathit{stretch}_4$ through function composition[1], propagated in above_4, and finally applied to wire connections in fan_4. An auxiliary definition *lzw* (stands for "long zip with") zips two lists by applying the binary operator to elements of the same index, and appending the remaining elements from the longer list to the resulting list. By calling *layout* on a circuit and supplying an identity function as the initial value of the accumulating parameter, we will get the layout.

Context-sensitive interpretations in Scala Context-sensitive interpretations in our OO approach are unproblematic as well:

```
trait Circuit4 extends Circuit1 { def layout(f: Int ⇒ Int): List[List[(Int, Int)]] }
trait Id4 extends Id1 with Circuit4 { def layout(f: Int ⇒ Int) = List() }
trait Fan4 extends Fan1 with Circuit4 {
  def layout(f: Int ⇒ Int) = List(for (i ← List.range(1, n)) yield (f(0), f(i)))
}
trait Above4 extends Above1 with Circuit4 {
  override val c1, c2: Circuit4
  def layout(f: Int ⇒ Int) = c1.layout(f) ++ c2.layout(f)
}
trait Beside4 extends Beside1 with Circuit4 {
  override val c1, c2: Circuit4
  def layout(f: Int ⇒ Int) = lzw(c1.layout(f), c2.layout(f.compose(c1.width + _)))(_ ++ _)
```

[1] A minor remark is that the composition order for f is incorrect in Gibbons and Wu's paper.

```
}
trait Stretch4 extends Stretch1 with Circuit4 {
  override val c : Circuit4
  def layout (f : Int ⇒ Int) = {
    val vs = ns.scanLeft (0) (_ + _).tail
    c.layout (f.compose (vs (_) − 1))
  }
}
def lzw [A] (xs : List [A], ys : List [A]) (f : (A, A) ⇒ A) : List [A] = (xs, ys) match {
  case (Nil, _)          ⇒ ys
  case (_, Nil)          ⇒ xs
  case (x :: xs, y :: ys) ⇒ f (x, y) :: lzw (xs, ys) (f)
}
```

The Scala version is both modular and arguably more intuitive, since contexts are captured as method arguments. The implementation of *layout* is a direct translation from the Haskell version. There are some minor syntax differences that need explanations. Firstly, in Fan_4, a **for** *comprehension* is used for producing a list of connections. Secondly, for simplicity, anonymous functions are created without a parameter list. For example, inside $Beside_4$, $c_1.width + \_$ is a shorthand for $i \Rightarrow c_1.width + i$, where the placeholder _ plays the role of the named parameter i. Thirdly, function composition is achieved through the *compose* method defined on function values, which has a reverse composition order as opposed to $\circ$ in Haskell. Fourthly, *lzw* is implemented as a *curried function*, where the binary operator f is moved to the end as a separate parameter list for facilitating type inference.

3.4 Modular language constructs

Besides new interpretations, new language constructs may be needed when a DSL evolves. For example, in the case of SCANS, we may want a *rstretch* (right stretch) combinator which is similar to the *stretch* combinator but stretches a circuit oppositely.

New constructs in Haskell Shallow embeddings make the addition of *rstretch* easy by defining a new function:

```
rstretch       :: [Int] → Circuit4 → Circuit4
rstretch ns c = stretch4 (1 : init ns) c `beside4` id4 (last ns − 1)
```

rstretch happens to be syntactic sugar over existing constructs. For non-sugar constructs, a new function that implements all supported interpretations is needed.

New constructs in Scala Such simplicity of adding new constructs is retained in Scala. Differently from the Haskell approach, there is a clear distinction between syntactic sugar and ordinary constructs in Scala.

In Scala, a syntatic sugar is defined as a smart constructor upon other smart constructors:

```
def rstretch (ns : List [Int], c : Circuit4) = stretch (1 :: ns.init, beside (c, id (ns.last − 1)))
```

On the other hand, adding ordinary constructs is done by defining a new trait that implements $Circuit_4$. If we treated *rstretch* as an ordinary construct, its definition would be:

```
trait RStretch extends Stretch4 {
  override def layout(f : Int ⇒ Int) = {
    val vs = ns.scanLeft(ns.last − 1)(_ + _).init
    c.layout(f.compose(vs(_)))
  }
}
```

Such an implementation of *RStretch* illustrates another strength of our OO approach regarding modularity. Note that *RStretch* does not implement $Circuit_4$ directly. Instead, it inherits $Stretch_4$ and overrides the *layout* definition so as to reuse other interpretations as well as field declarations from $Stretch_4$. Inheritance and method overriding enable partial reuse of an existing language construct implementation, which is particularly useful for defining specialized constructs. However, such partial reuse is hard to achieve in Haskell.

3.5 Discussion

Gibbons and Wu claim that in shallow embeddings new language constructs are easy to add, but new interpretations are hard. It is possible to define multiple interpretations via tuples, "*but this is still a bit clumsy: it entails revising existing code each time a new interpretation is added, and wide tuples generally lack good language support*" (Gibbons & Wu, 2014). In other words, Haskell's approach based on tuples is essentially non-modular. However, as our OOP approach shows, in OOP both language constructs and interpretations are easy to add in shallow embeddings. In other words, the circuit DSL presented so far does not suffer from the Expression Problem. The key point is that procedural abstraction combined with OOP features (subtyping, inheritance, and type-refinement) adds expressiveness over traditional procedural abstraction.

Gibbons and Wu do discuss a number of advanced techniques (Carette *et al.*, 2009; Swierstra, 2008) that can solve *some* of the modularity problems. For example, Carette *et al*. (2009) deal with multiple interpretations (Section 3.1) using type classes. However, while useful (see also Section 4), these techniques complicate the encoding of the EDSL. Moreover, dependent interpretations (Section 3.2) remain non-modular because an encoding via tuples is still needed. In contrast, the approach proposed here is just straightforward OOP, it uses only simple types, and dependent interpretations are not a problem.

4 Modular terms

One potential criticism to the approach presented so far is that while the interpretations are modular, building terms is not. Every time we develop new interpretations, a new set of companion smart constructors has to be developed as well. Unfortunately the different smart constructors build terms that are specific to a particular interpretation, leading to duplication of code whenever the same term needs to be run with different interpretations. Fortunately, there is an easy solution to this problem: we overload the constructors, making them independent of any specific interpretation.

Abstract factories To capture the generic interface of the constructors we use an abstract factory for circuits:

```
trait Factory[Circuit] {
  def id(x: Int): Circuit
  def fan(x: Int): Circuit
  def above(x: Circuit, y: Circuit): Circuit
  def beside(x: Circuit, y: Circuit): Circuit
  def stretch(x: Circuit, xs: Int*): Circuit
}
```

Factory is a generic interface, which exposes factory methods for each circuit construct supported by SCANS. The idea of capturing the interfaces of constructors is inspired by the Finally Tagless (Carette *et al.*, 2009) or Object Algebras (Oliveira & Cook, 2012) approaches, which employ such a technique.

Abstract terms Modular terms can be constructed via the abstract factory. For example, the circuit shown in Fig. 2 is built as:

```
def c[Circuit](f: Factory[Circuit]) =
  f.above(f.beside(f.fan(2), f.fan(2)),
          f.above(f.stretch(f.fan(2), 2, 2),
                  f.beside(f.beside(f.id(1), f.fan(2)), f.id(1))))
```

c is a generic method that takes an *Factory* instance and builds a circuit through that instance. With Scala the definition of *c* can be simpler: we can avoid prefixing "*f*." everywhere by importing *f*. Nevertheless, the definition shown here is more language-independent.

Concrete factories We need concrete factories that implement *Factory* to actually invoke *c*. Here is a concrete factory that produces $Circuit_1$:

```
trait Factory_1 extends Factory[Circuit_1] {...}
```

where the omitted code is identical to the smart constructors presented in Section 2.2. Concrete factories for other circuit implementations can be defined in a similar way by instantiating the type parameter *Circuit* accordingly:

```
trait Factory_4 extends Factory[Circuit_4] {...}
```

Concrete terms Supplying concrete factories to abstract terms, we obtain concrete terms that can be interpreted differently:

```
c(new Factory_1 { }).width//4
c(new Factory_4 { }).layout {x => x}//List(List((0,1),(2,3)), List((1,3)), List((1,2)))
```

Modular extensions Both factories and terms can be *modularly* reused when the DSL is extended with new language constructs. To support right stretch for SCANS, we first extend the abstract factory with new factory methods:

```
trait ExtendedFactory[Circuit] extends Factory[Circuit] {
  def rstretch(x: Circuit, xs: Int*): Circuit
}
```

We can also build extended concrete factories upon existing concrete factories:

$$\begin{array}{l}\textbf{trait}\ \mathit{ExtendedFactory}_4\ \textbf{extends}\ \mathit{ExtendedFactory}[\mathit{Circuit}_4]\ \textbf{with}\ \mathit{Factory}_4\ \{\\ \quad \textbf{def}\ \mathit{rstretch}(x:\mathit{Circuit}_4, xs:\mathit{Int}*) = \textbf{new}\ \mathit{RStretch}\ \{\textbf{val}\ c = x; \textbf{val}\ ns = xs.\mathit{toList}\}\\ \}\end{array}$$

Furthermore, previously defined terms can be reused in constructing extended terms:

$$\textbf{def}\ c_2[\mathit{Circuit}](f:\mathit{ExtendedFactory}[\mathit{Circuit}]) = f.\mathit{rstretch}(c(f),2,2,2,2)$$

5 Case study: A shallow EDSL for SQL queries

A common motivation for using deep embeddings is performance. Deep embeddings enable complex transformations to be defined over the AST, which is useful to implement optimizations that improve the performance. An alternative way to obtain performance is to use staging frameworks, such as Lightweight Modular Staging (LMS) (Rompf & Odersky, 2010). As illustrated by Rompf and Amin (2015) staging can preclude the need for manual optimizations based on user-defined transformations over the AST for a realistic query DSL. To further illustrate the applicability of shallow OO embeddings, we refactored Rompf and Amin's deep, external DSL implementation to make it more *modular*, *shallow* and *embedded*. The shallow DSL retains the performance of the original deep DSL.

5.1 Overview

SQL is the best-known DSL for data queries. Rompf and Amin (2015) present a SQL query processor implementation in Scala. Their implementation is an *external* DSL, which first parses a SQL query into a relational algebra AST and then executes the query in terms of that AST. Based on the LMS framework (Rompf & Odersky, 2010), the SQL compilers are nearly as simple as an interpreter while having performance comparable to hand-written code. The implementation uses deep embedding techniques such as algebraic datatypes (*case classes* in Scala) and pattern matching for representing and interpreting ASTs. These techniques are a natural choice as multiple interpretations are needed for supporting different backends. But problems arise when the implementation evolves with new language constructs. All existing interpretations have to be modified for dealing with these new cases, suffering from the Expression Problem.

We refactored Rompf and Amin (2015)'s implementation into a shallow EDSL for the following reasons. Firstly, multiple interpretations are no longer a problem for our shallow OO embedding technique. Secondly, the original implementation contains no hand-coded transformations over AST, due to the use of staging. Thirdly, it is common to embed SQL into a general purpose language.

To illustrate our shallow EDSL, suppose that there is a data file *talks.csv* that contains a list of talks with time, title and room. We can write several sample queries on this file with our EDSL. A simple query that lists all items in *talks.csv* is:

$$\textbf{def}\ q_0 = \mathit{FROM}(\texttt{"talks.csv"})$$

Another query that finds all talks at 9 am with their room and title selected is:

def $q_1 = q_0$ *WHERE* $'time ===$ `"09:00 AM"` *SELECT* $('room, 'title)$

Yet another relatively complex query to find all unique talks happening at the same time in the same room is:

def $q_2 = q_0$ *SELECT* $('time, 'room, 'title$ *AS* $'title_1)$ *JOIN*
$(q_0$ *SELECT* $('time, 'room, 'title$ *AS* $'title_2))$ *WHERE*
$'title_1 <> 'title_2$

Compared to an external implementation, our embedded implementation has the benefit of reusing the mechanisms provided by the host language for free. As illustrated by the sample queries above, we are able to reuse common subqueries (q_0) in building complex queries (q_1 and q_2). This improves the readability and modularity of the embedded programs.

5.2 Embedded syntax

Thanks to the good support for EDSLs in Scala, we can precisely model the syntax of SQL. The syntax of our EDSL is close to that of LINQ (Meijer *et al.*, 2006), where **select** is an optional, terminating the clause of a query. We employ well-established OO and Scala techniques to simulate the syntax of SQL queries in our shallow EDSL implementation. Specifically, we use the *Pimp My Library* pattern (Odersky, 2006) for lifting field names and literals implicitly. For the syntax of combinators such as **where** and **join**, we adopt a fluent interface style. Fluent interfaces enable writing something like "*FROM*$(...)$.*WHERE*$(...)$.*SELECT*$(...)$". Scala's infix notation further omits "." in method chains. Other famous embedded SQL implementations in OOP such as LINQ (Meijer *et al.*, 2006) also adopt similar techniques in designing their syntax. The syntax is implemented in a pluggable way, in the sense that the semantics is decoupled from the syntax. Details of the syntax implementation are beyond the scope of this pearl. The interested reader can consult the companion code.

Beneath the surface syntax, a relational algebra operator structure is constructed. For example, we will get the following operator structure for q_1:

Project(*Schema*(`"room"`, `"title"`),
Filter(*Eq*(*Field*(`"time"`), *Value*(`"09:00 AM"`)),
Scan(`"talks.csv"`)))

5.3 A relational algebra compiler

A SQL query can be represented by a relational algebra operator. The basic interface of operators is modeled as follows:

```
trait Operator {
  def resultSchema: Schema
  def execOp(yld: Record ⇒ Unit): Unit
}
```

Two interpretations, *resultSchema* and *execOp*, need to be implemented for each concrete operator: the former collects a schema for projection; the latter executes actions to the

records of the table. Very much like the interpretation *layout* discussed in Section 3.3, *execOp* is both *context-sensitive* and *dependent*: it takes a callback *yld* and accumulates what the operator does to records into *yld* and uses *resultSchema* in displaying execution results. Here are some core concrete relational algebra operators:

```
trait Project extends Operator {
  val out, in : Schema; val op : Operator
  def resultSchema = out
  def execOp (yld : Record ⇒ Unit) = op.execOp { rec ⇒ yld (Record (rec (in ), out)) }
}
trait Join extends Operator {
  val op1, op2 : Operator
  def resultSchema = op1.resultSchema ++ op2.resultSchema
  def execOp (yld : Record ⇒ Unit) =
    op1.execOp { rec1 ⇒
      op2.execOp { rec2 ⇒
        val keys = rec1.schema intersect rec2.schema
        if (rec1 (keys) ≡ rec2 (keys))
          yld (Record (rec1.fields ++ rec2.fields, rec1.schema ++ rec2.schema)) } }
}
trait Filter extends Operator {
  val pred : Predicate; val op : Operator
  def resultSchema = op.resultSchema
  def execOp (yld : Record ⇒ Unit) = op.execOp { rec ⇒ if (pred.eval (rec)) yld (rec) }
}
```

Project rearranges the fields of a record; *Join* matches a record against another and combines the two records if their common fields share the same values; *Filter* keeps a record only when it meets a certain predicate. There are also two utility operators, *Print* and *Scan*, for processing inputs and outputs, whose definitions are omitted for space reasons.

From an interpreter to a compiler The query interpreter presented so far is elegant but unfortunately slow. To achieve better performance, Rompf and Amin extend the SQL processor in various ways. The idea is to turn the slow query interpreter into a fast query compiler by generating specialized low-level code for a given query. With the help of the LMS framework, this task becomes rather easy. LMS provides a type constructor *Rep* for annotating computations that are to be performed in the next stage. The signature of the staged *execOp* is:

```
def execOp (yld : Record ⇒ Rep [Unit]) : Rep [Unit]
```

where *Unit* is lifted as *Rep* [*Unit*] for delaying the actions on records to the generated code. Two staged versions of *execOp* are introduced for generating Scala and C code respectively. By using the technique presented in Section 3, they are added modularly with existing interpretations such as *resultSchema* reused. The implementation of staged *execOp* is almost identical to the unstaged counterpart except for minor API differences on staged

and unstaged types. Hence the simplicity of the implementation remains. At the same time, dramatic speedups are obtained by switching from interpretation to compilation.

Language extensions Rompf and Amin also extend the query processor with two new language constructs, hash joins and aggregates. Differently from the original implementation, the introduction of these constructs is done in a modular manner with our shallow OO embedding:

```
trait Group extends Operator {
  val keys, agg : Schema; val op : Operator
  def resultSchema = keys ++ agg
  def execOp (yld : Record ⇒ Unit) {...}
}
trait HashJoin extends Join {
  override def execOp (yld : Record ⇒ Unit) = {
    val keys = op1.resultSchema intersect op2.resultSchema
    val hm = new HashMapBuffer (keys, op1.resultSchema)
    op1.execOp {rec1 ⇒
      hm (rec1 (keys)) += rec1.fields}
    op2.execOp {rec2 ⇒
      hm (rec2 (keys)) foreach {rec1 ⇒
        yld (Record (rec1.fields ++ rec2.fields, rec1.schema ++ rec2.schema))}}}
}
```

Group supports SQL's **group by** clause, which partitions records and sums up specified fields from the composed operator. *HashJoin* is a replacement for *Join*, which uses an hash-based implementation instead of naive nested loops. With inheritance and method overriding, we are able to reuse the field declarations and other interpretations from *Join*.

5.4 Evaluation

We evaluate our refactored shallow implementation with respect to the original deep implementation. Both implementations of the DSL (the original and our refactored version) *generate the same code*: thus the performance of the two implementations is similar. Hence we compare the two implementations only in terms of the source lines of code (SLOC). To make the comparison fair, only the code for the interpretations are compared (code related to surface syntax is excluded). The refactored version takes only a few more lines than the original version, as seen in the table.

Source	Functionality	Deep	Shallow
query_unstaged	SQL interpreter	83	95
query_staged	SQL to Scala compiler	179	191
query_optc	SQL to C compiler	245	259

Table 1: *SLOC for original (Deep) and refactored (Shallow) versions.*

6 Conclusion

This functional pearl revealed the close correspondence between OOP and shallow embeddings - the essence of both is procedural abstraction. It also showed how OOP increases the modularity of shallow EDSLs. OOP abstractions, including subtyping, inheritance, and type-refinement, bring extra modularity to traditional procedural abstraction. As a result, multiple interpretations are allowed to co-exist in shallow embeddings.

It has always been a hard choice between shallow and deep embeddings when designing an EDSL: there are some tradeoffs between the two styles. A nice aspect of shallow embeddings is their simplicity. Deep embeddings trade some simplicity and the ability to add new language constructs for some extra power. This extra power enables multiple interpretations, as well as complex transformations over the AST. As this pearl shows, in OO languages, multiple interpretations are still easy to do with shallow embeddings. Moreover, Erwig and Walkingshaw (2014) argue that shallow embeddings are more semantics-driven compared to deep embeddings which have many preferable properties in designing DSLs. Therefore the motivation to employ deep embeddings becomes weaker. Nevertheless, the need for transformations over the AST is still a valid reason to switch to deep embeddings.

References

Carette, Jacques, Kiselyov, Oleg, & Shan, Chung-chieh. (2009). Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages. *JFP*, **19**(05), 509–543.

Cook, William R. (2009). On Understanding Data Abstraction, Revisited. *In OOPSLA*.

Erwig, Martin, & Walkingshaw, Eric. (2014). Semantics-Driven DSL Design. *Computational Linguistics: Concepts, Methodologies, Tools, and Applications*.

Gibbons, Jeremy, & Wu, Nicolas. (2014). Folding Domain-Specific Languages: Deep and Shallow Embeddings (Functional Pearl). *In ICFP*.

Hinze, Ralf. (2004). An Algebra of Scans. *In MPC*.

Hudak, Paul. (1998). Modular Domain Specific Languages and Tools. *In ICSR*.

Jovanovic, Vojin, Shaikhha, Amir, Stucki, Sandro, Nikolaev, Vladimir, Koch, Christoph, & Odersky, Martin. (2014). Yin-Yang: Concealing the Deep Embedding of DSLs. *In GPCE*.

Meijer, Erik, Beckman, Brian, & Bierman, Gavin. (2006). LINQ: Reconciling Object, Relations and XML in the .NET Framework. *In SIGMOD/PODS*.

Odersky, Martin. (2006). *Pimp My Library*.

Oliveira, Bruno C. d. S., & Cook, William R. (2012). Extensibility for the Masses. *In ECOOP*.

Reynolds, John C. (1975). User-defined Types and Procedural Data Structures as Complementary Approaches to Type Abstraction. *New Directions in Algorithmic Languages*.

Rompf, Tiark, & Amin, Nada. (2015). Functional Pearl: A SQL to C Compiler in 500 Lines of Code. *In ICFP*.

Rompf, Tiark, & Odersky, Martin. (2010). Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. *In GPCE*.

Svenningsson, Josef, & Axelsson, Emil. (2012). Combining Deep and Shallow Embedding for EDSL. *In TFP*.

Swierstra, Wouter. (2008). Data Types à la Carte. *JFP*, **18**(04), 423–436.

Wadler, Philip. 1998 (Nov.). *The Expression Problem*. Note to Java Genericity mailing list.

Wang, Yanlin, & Oliveira, Bruno C. d. S. (2016). The Expression Problem, Trivially! *In Modularity*.