Folding Domain-Specific Languages: Deep and Shallow Embeddings

(Functional Pearl)

Jeremy Gibbons  Nicolas Wu
Department of Computer Science, University of Oxford
{jeremy.gibbons,nicolas.wu}@cs.ox.ac.uk

Abstract

A domain-specific language can be implemented by embedding within a general-purpose host language. This embedding may be deep or shallow, depending on whether terms in the language construct syntactic or semantic representations. The deep and shallow styles are closely related, and intimately connected to folds; in this paper, we explore that connection.

1. Introduction

General-purpose programming languages (GPLs) are great for generality. But this very generality can count against them: it may take a lot of programming to establish a suitable context for a particular domain; and the programmer may end up being spoilt for choice with the options available to her—especially if she is a domain specialist rather than primarily a software engineer. This tension motivates many years of work on techniques to support the development of domain-specific languages (DSLs) such as VHDL, SQL and PostScript: languages specialized for a particular domain, incorporating the contextual assumptions of that domain and guiding the programmer specifically towards programs suitable for that domain.

There are two main approaches to DSLs. Standalone DSLs provide their own custom syntax and semantics, and standard compilation techniques are used to translate or interpret programs written in the DSL for execution. Standalone DSLs can be designed for maximal convenience to their intended users. But the exercise can be a significant undertaking for the implementer, involving an entirely separate ecosystem—compiler, editor, debugger, and so on—and typically also much reinvention of standard language features such as local definitions, conditionals, and iteration.

The alternative approach is to embed the DSL within a host GPL, essentially as a collection of definitions written in the host language. All the existing facilities and infrastructure of the host environment can be appropriated for the DSL, and familiarity with the syntactic conventions and tools of the host language can be carried over to the DSL. Whereas the standalone approach is the most common one within object-oriented circles [11], the embedded approach is typically favoured by functional programmers [19]. It seems that core FP features such as algebraic datatypes and higher-order functions are extremely helpful in defining embedded DSLs; conversely, it has been said [24] that language-oriented tasks such as DSLs are the killer application for FP.

Amongst embedded DSLs, there are two further refinements. With a deep embedding, terms in the DSL are implemented simply to construct an abstract syntax tree (AST), which is subsequently transformed for optimization and traversed for evaluation. With a shallow embedding, terms in the DSL are implemented directly by their semantics, bypassing the intermediate AST and its traversal. The names ‘deep’ and ‘shallow’ seem to have originated in the work of Boulton and colleagues on embedding hardware description languages in theorem provers for the purposes of verification [6]. Boulton’s motivation for the names was that a deep embedding preserves the syntactic representation of a term, “whereas in a shallow embedding [the syntax] is just a surface layer that is easily blown away by rewriting” [5]. It turns out that deep and shallow embeddings are closely related, and intimately connected to folds; our purpose in this paper is to explore that connection.

2. Embedding DSLs

We start by looking a little closer at deep and shallow embeddings. Consider a very simple language of arithmetic expressions, involving integer constants and addition:

\[
\text{type } \text{Expr} = \ldots
\]

\[
\text{lit :: Integer } \rightarrow \text{Expr}
\]

\[
\text{add :: Expr} \rightarrow \text{Expr} \rightarrow \text{Expr}
\]

The expression \((3 + 4) + 5\) is represented in the DSL by the term \(\text{add (add (lit 3) (lit 4)) (lit 5)}\).

As a deeply embedded DSL, the two operations \text{lit} and \text{add} are encoded directly as constructors of an algebraic datatype:

\[
\text{data Expr} :: \times \text{where}
\]

\[
\text{Lit :: Integer } \rightarrow \text{Expr}
\]

\[
\text{Add :: Expr} \rightarrow \text{Expr} \rightarrow \text{Expr}
\]

\[
\text{lit n } = \text{Lit n}
\]

\[
\text{add x y } = \text{Add x y}
\]

(We have used Haskell’s ‘generalized algebraic datatype’ notation, in order to make the types of the constructors \text{Lit} and \text{Add} explicit; but we are not using the generality of GADTs here, and the old-fashioned way would have worked too.) Observations of terms in the DSL are defined as functions over the algebraic datatype. For example, here is how to evaluate an expression:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICFP '14  September 1–6, 2014, Gothenburg, Sweden.
Copyright © 2014 ACM 978-1-4503-2873-9/14/09...$15.00.
http://dx.doi.org/10.1145/2628136.2628138
We have used subscripts to distinguish different representations of $\text{Expr}$ with those of $\text{Expr}_2$. This might be used as follows:

\[
\text{eval}_2 :: \text{Expr}_2 \rightarrow \text{Integer} \\
\text{eval}_2 (\text{Lit } n) = n \\
\text{eval}_2 (\text{Add } x \ y) = \text{eval}_2 x + \text{eval}_2 y
\]

This might be used as follows:

\[
> \text{eval}_2 (\text{Add } (\text{Add } (\text{Lit } 3) (\text{Lit } 4)) (\text{Lit } 5))
\]

12

In other words, a deep embedding consists of a representation of the abstract syntax as an algebraic datatype, together with some functions that assign semantics to that syntax by traversing the algebraic datatype.

A shallow embedding eschews the algebraic datatype, and hence the explicit representation of the abstract syntax of the language; instead, the language is defined directly in terms of its semantics. For example, if the semantics is to be evaluation, then we could define:

\[
\text{type } \text{Expr}_3 = \text{Integer} \\
\text{lit } n = n \\
\text{add } x \ y = x + y \\
\text{eval}_3 :: \text{Expr}_3 \rightarrow \text{Integer} \\
\text{eval}_3 n = n
\]

This might be used as follows:

\[
> \text{eval}_3 (\text{add } (\text{add } (\text{lit } 3) (\text{lit } 4)) (\text{lit } 5))
\]

12

We have used subscripts to distinguish different representations of morally ‘the same’ functions ($\text{eval}_2$ and $\text{eval}_3$) and types ($\text{Expr}_2$ and $\text{Expr}_3$). We will continue that convention throughout the paper.

One might see the deep and shallow embeddings as duals, in a variety of senses. For one sense, the language constructs $\text{Lit}$ and $\text{Add}$ in the deep embedding do none of the work, leaving this entirely to the observation function $\text{eval}$; in contrast, in the shallow embedding, the language constructs $\text{lit}$ and $\text{add}$ do all the work, and the observer $\text{eval}_3$ is simply the identity function.

For a second sense, it is trivial to add a second observer such as pretty-printing to the deep embedding—just define another function alongside $\text{eval}$—but awkward to add a new construct such as multiplication: doing so entails revisiting the definitions of all existing observers to add an additional clause. In contrast, adding a construct to the shallow embedding—alongside $\text{lit}$ and $\text{add}$—is trivial, but the obvious way of introducing an additional observer entails completely revising the semantics by changing the definitions of all existing constructs. This is precisely the tension underlying the expression problem [23, 30], so named for precisely this example.

The types of $\text{lit}$ and $\text{add}$ in the shallow embedding coincide with those of $\text{Lit}$ and $\text{Add}$ in the deep embedding; moreover, the definitions of $\text{lit}$ and $\text{add}$ in the shallow embedding correspond to the ‘actions’ in each clause of the definition of the observer in the deep embedding. The shallow embedding presents a compositional semantics for the language, since the semantics of a composite term is explicitly composed from the semantics of its components. Indeed, it is only such compositional semantics that can be captured in a shallow embedding; it is possible to define a more sophisticated non-compositional semantics as an interpretation of a deep embedding, but not possible to represent that semantics directly via a shallow embedding.

However, there is no duality in the categorical sense of reversing arrows. Although deep and shallow embeddings have been called the ‘initial’ and ‘final’ approaches [8], in fact the two approaches are equivalent, and both correspond to initial algebras; Carette et al. say only that they use the term ‘final’ “because we represent each object term not by its abstract syntax but by its denotation in a semantic algebra”, and they are not concerned with final coalgebras.

3. Scans

The expression language above is very simple—perhaps too simple to serve as a convincing vehicle for discussion. As a more interesting example of a DSL, we turn to a language for parallel prefix circuits [14], which crop up in a number of different applications—carry-lookahead adders, parallel sorting, and stream compaction, to name but a few. Given an associative binary operator $\circ$, a prefix computation of width $n > 0$ takes a sequence $x_1, x_2, \ldots, x_n$ of inputs and produces the sequence $x_1 \circ x_2 \circ \cdots \circ x_n$ of outputs. A parallel prefix circuit performs this computation in parallel, in a fixed format independent of the input values $x_i$.

An example of such a circuit is depicted in Figure 1. This circuit diagram should be read as follows. The inputs are fed in at the top, and the outputs fall out at the bottom. Each node (the blobs in the diagram) represents a local computation, combining the values on each of its input wires using $\circ$, in left-to-right order, and providing copies of the result on each of its output wires. It is an instructive exercise to check that this circuit does indeed take $x_1, x_2, x_3$ to $x_1 \circ x_2 \circ x_3$.

Such circuits can be constructed using the following operators:

\[
\text{type Size } = \text{Int} \rightarrow \text{Positive} \\
\text{type } \text{Circuit}_1 = \ldots \\
\text{id} :: \text{Size } \rightarrow \text{Circuit}_1 \\
\text{fan } :: \text{Size } \rightarrow \text{Circuit}_1 \\
\text{above } :: \text{Circuit}_1 \rightarrow \text{Circuit}_1 \rightarrow \text{Circuit}_1 \\
\text{aside } :: \text{Circuit}_1 \rightarrow \text{Circuit}_1 \rightarrow \text{Circuit}_1 \\
\text{stretch } :: [\text{Size}] \rightarrow \text{Circuit}_1 \rightarrow \text{Circuit}_1
\]

The most basic building block is the identity circuit, $\text{id } n$, which creates a circuit consisting of $n$ parallel wires that copy input to output. The other primitive is the fan circuit; $\text{fan } n$ takes $n$ inputs, and adds its first input to each of the others. We only consider non-empty circuits, so $n$ must be positive in both cases. Instances of $\text{id}$ and $\text{fan}$ of width 4 are shown in Figure 2.

Then there are three combinators for circuits. The series or vertical composition, $\text{above } c d$, takes two circuits $c$ and $d$ of the same width, and connects the outputs of $c$ to the inputs of $d$. The parallel or horizontal composition, $\text{aside } c d$, places $c$ beside $d$, leaving them unconnected; there are no width constraints on $c$ and $d$.

Figure 3 shows a 2-fan beside a 1-identity, a 1-identity beside a 2-fan, and the first of these above the second (note that they both have width 3); this yields the “serial” parallel prefix circuit of width 3.

Finally, the stretch combinator, $\text{stretch } ws c$, takes a non-empty list of positive widths $w_1, \ldots, w_n$ of length $n$, and a circuit $c$ of width $n$, and “stretches” $c$ out to width $\text{sum } w$ by interleaving some additional wires. Of the first bundle of $w_1$ inputs, the last is routed to the first input of $c$ and the rest pass straight through; of
the next bundle of \( w_2 \) inputs, the last is routed to the second input of \( c \) and the rest pass straight through; and so on. (Note that each bundle width \( w_i \) must be positive.) For example, Figure 4 shows a 3-fan stretched out to width 8, in bundles of \([3,2,3]\).

So one possible construction of the Brent–Kung parallel prefix circuit in Figure 1 is

\[
\text{fan } 2 \ '\text{ beside' } \text{ fan } 2 \ '\text{ above' } \\
\text{stretch } [2,2] \ (\text{fan } 2) \ '\text{ above' } \\
(\text{id} \ '\text{ beside' } \text{ fan } 2 \ '\text{ beside' } \text{ identity' } 1)
\]

The general Brent–Kung construction [7] is given recursively. The general pattern is a row of 2-fans, possibly with an extra wire in the case of odd width; then a Brent–Kung circuit of half the width, stretched out by a factor of two; then another row of 2-fans, shifted one place to the right.

\[
brentkung :: \text{Size} \rightarrow \text{Circuit}_1 \\
brentkung 1 = \text{id} \rightarrow 1 \\
brentkung w \\
\quad = (\text{row} \ (\text{replicate} \ u \ (\text{fan } 2)) \ '\text{ pad' } w \ '\text{ above' } \\
\quad \ (\text{stretch} \ (\text{replicate} \ u \ 2) \ (\text{brentkung} \ u) \ '\text{ pad' } w \ '\text{ above' } \\
\quad \ (\text{row} \ (\text{id} \ '\text{ replicate' } v \ (\text{fan } 2)) \ '\text{ pad' } (w - 1)) \\
\quad \text{where} \ (u,v) = (w '\text{ div' } 2, (w - 1) '\text{ div' } 2) \\
\quad \ '\text{ pad' } w = \text{if even } w \ \text{then} \ '\text{ else' } c '\text{ beside' } \text{ identity' } 1 \\
\quad \text{row} = \text{foldr1} \ \text{ beside }
\]

The Brent–Kung circuit of width 16 is shown in Figure 5. Note one major benefit of defining \text{Circuit} as an embedded DSL: we can exploit for free host language constructions such as \text{replicate} and \text{foldr1}, rather than having to reinvent them within the DSL.

As a deeply embedded DSL, circuits can be captured by the following algebraic datatype:

\[
\text{data Circuit}_2 :: \ast \ \text{ where} \\
\quad \text{Identity} :: \text{Size} \rightarrow \text{Circuit}_2 \\
\quad \text{Fan} :: \text{Size} \rightarrow \text{Circuit}_2 \\
\quad \text{Above} :: \text{Circuit}_2 \rightarrow \text{Circuit}_2 \rightarrow \text{Circuit}_2 \\
\quad \text{Beside} :: \text{Circuit}_2 \rightarrow \text{Circuit}_2 \rightarrow \text{Circuit}_2 \\
\quad \text{Stretch} :: \text{Size} \rightarrow \text{Circuit}_2 \rightarrow \text{Circuit}_2
\]

It is, of course, straightforward to define functions to manipulate this representation. Here is one, which computes the width of a circuit:

\[
\text{type Width} = \text{Int} \\
\text{width2 :: Circuit}_2 \rightarrow \text{Width} \\
\text{width2 \ (Identity \ w) } = w \\
\text{width2 \ (Fan \ w) } = w \\
\text{width2 \ (Above \ x \ y) } = \text{width2} \ x \\
\text{width2 \ (Beside \ x \ y) } = \text{width2} \ x + \text{width2} \ y \\
\text{width2 \ (Stretch \ ws \ x) } = \text{sum} \ ws
\]

Note that \text{width2} is compositional: it is a fold over the abstract syntax of \text{Circuit}_2. That makes it a suitable semantics for a shallow embedding. That is, we could represent circuits directly by their widths, as follows:

\[
\text{type Circuit}_3 = \text{Width} \\
\text{id} \ \text{entity \ w } = w
\]

Clearly, width is a rather uninteresting semantics to give to circuits. But what other kinds of semantics will fit the pattern of compositionality, and so be suitable for a shallow embedding? In order to explore that question, we need to look a bit more closely at folds and their variations.

4. Folds

Folds are the natural pattern of computation induced by inductively defined algebraic datatypes. We consider here just polynomial algebraic datatypes, namely those with one or more constructors, each constructor taking zero or more arguments to the datatype being defined, and each argument either having a fixed type independent of the datatype, or being a recursive occurrence of the datatype itself. For example, the polynomial algebraic datatype \text{Circuit}_2 above has five constructors; \text{Identity} and \text{Fan} each take one argument of the fixed type \text{Size}; \text{Above} and \text{Beside} take two arguments, both recursive occurrences; \text{Stretch} takes two arguments, one of which is the fixed type \text{Size}, and the other is a recursive argument. Thus, we rule out contravariant recursion, polymorphic datatypes, higher kinds, and other such esoterica. For simplicity, we also ignore DSLs with binding constructs, which complicate matters significantly; for more on this, see [1, 8].

The general case is captured by a shape—also called a base or pattern functor—which is an instance of the \text{Functor} type class:

\[
\text{class Functor } f \ \text{ where} \\
\quad \text{fmap} :: \ (a \rightarrow b) \rightarrow (f \ a \rightarrow f \ b)
\]

For \text{Circuit}_2, the shape is given by \text{CircuitF} as follows, where the parameter \( x \) marks the recursive spots:

\[
\text{data CircuitF} :: \ast \rightarrow \ast \ \text{ where} \\
\quad \text{IdentityF} :: \text{Size} \rightarrow \text{CircuitF} \ x \\
\quad \text{FanF} :: \text{Size} \rightarrow \text{CircuitF} \ x \\
\quad \text{AboveF} :: \text{x} \rightarrow \text{x} \rightarrow \text{CircuitF} \ x \\
\quad \text{BesideF} :: \text{x} \rightarrow \text{x} \rightarrow \text{CircuitF} \ x \\
\quad \text{StretchF} :: \text{[Size]} \rightarrow \text{x} \rightarrow \text{CircuitF} \ x
\]

\[
\text{instance Functor CircuitF} \ \text{ where} \\
\quad \text{fmap} \ (\text{IdentityF} \ w) = \text{IdentityF} \ w \\
\quad \text{fmap} \ (\text{FanF} \ w) = \text{FanF} \ w \\
\quad \text{fmap} \ (\text{AboveF} \ x \ y) = \text{AboveF} \ (f \ x) \ (f \ y) \\
\quad \text{fmap} \ (\text{BesideF} \ x \ y) = \text{BesideF} \ (f \ x) \ (f \ y) \\
\quad \text{fmap} \ (\text{StretchF} \ ws \ x) = \text{StretchF} \ ws \ (f \ x)
\]

We can use this shape functor as the basis of an alternative definition of the algebraic datatype \text{Circuit}_2:

\[
\text{data Circuit}_4 = \text{In} \ (\text{CircuitF} \ \text{Circuit}_4)
\]

Now, an algebra for a functor \text{f} consists of a type \text{a} and a function taking an \text{f}-structure of \text{a}-values to an \text{a}-value. For the functor \text{CircuitF}, this is:

\[
\text{type CircuitAlg } a = \text{CircuitF} \ a \rightarrow a
\]
We discuss these consequences next.

As mentioned above, the deep embedding smoothly supports additional observations, rather than just the identity function.

Now the observation functions width and depth of a circuit? It's not much more difficult than with a deep embedding; we simply make the semantics finding both the width and the depth of a circuit. No problem—we can just define an interpretation mean that the interpretation of a whole may be determined solely from the interpretations of its parts; it is both a valuable property for reasoning and a significant limitation to expressivity. Not all interpretations are of this form; sometimes a 'primary' interpretation of the whole depends also on 'secondary' interpretations of its parts.

4.1 Multiple interpretations

As mentioned above, the deep embedding smoothly supports additional observations. For example, suppose that we also wanted to find the depth of our circuits. No problem—we can just define another observation function.

\[
\text{depthAlg :: CircuitAlg Depth} \\
\text{depthAlg (IdentityF w) = 0} \\
\text{depthAlg (FanF w) = 1} \\
\text{depthAlg (AboveF x y) = x + y} \\
\text{depthAlg (BesideF x y) = x * \text{'max'} \ y} \\
\]

But what about with a shallow embedding? With this approach, circuits can only have a single semantics, so how do we accommodate both the width and the depth of a circuit? It’s not much more difficult than with a deep embedding; we simply make the semantics a pair, providing both interpretations simultaneously.

\[
\text{type Depth = Int} \\
\text{depthAlg :: CircuitAlg Depth} \\
\text{depthAlg (IdentityF w) = 0} \\
\text{depthAlg (FanF w) = 1} \\
\text{depthAlg (AboveF x y) = x + y} \\
\text{depthAlg (BesideF x y) = x * \text{'max'} \ y} \\
\]

Now the observation functions width\(_5\) and depth\(_5\) become projections, rather than just the identity function.

For example, width is a fold for the deeply embedded DSL of shape Circuit\(_F\), and is determined by the following algebra:

\[
\begin{align*}
\text{foldC :: CircuitAlg a} & \rightarrow \text{Circuit}_4 \rightarrow a \\
\text{foldC h (In x) = h (fmap (foldC h) x)}
\end{align*}
\]

So a compositional observation function for the deep embedding, such as width\(_4\), is precisely a fold using such an the algebra. We know a lot about folds, and this tells us a lot about embedded DSLs. We discuss these consequences next.

4.2 Dependent interpretations

A shallow embedding supports only compositional interpretations, whereas a deep embedding provides full access to the AST and hence also non-compositional manipulations. Here, 'compositional' of an interpretation means that the interpretation of a whole may be determined solely from the interpretations of its parts; it is both a valuable property for reasoning and a significant limitation to expressivity. Not all interpretations are of this form; sometimes a 'primary' interpretation of the whole depends also on 'secondary' interpretations of its parts.

For example, whether a circuit is well formed depends on the widths of its constituent parts. Given that we have an untyped (or rather, 'unsized') model of circuits, we might capture this property in a separate function wellSized:

\[
\text{type WellSized = Bool} \\
\text{wellSized :: Circuit}_5 \rightarrow \text{WellSized} \\
\text{wellSized (Identity w) = True} \\
\text{wellSized (Fan w) = True}
\]
4.3 Context-sensitive interpretations

Consider generating a circuit layout from a circuit description, for example as the first step in expressing the circuit in a hardware description language such as VHDL—or, for that matter, for producing the diagrams in this paper. The essence of the translation is to determine the connections between vertical wires. Note that each circuit can be thought of as a sequence of layers, and connections only go from one layer to the next (and only rightwards, too). So it suffices to generate a list of layers, where each layer is a collection of pairs \((i, j)\) denoting a connection from wire \(i\) on this layer to wire \(j\) on the next. The ordering of the pairs on each layer is not significant. We count from 0. For example, the Brent–Kung circuit of size 4 given in Figure 1 has the following connections:

\[
((0,1), (2,3)), \quad [(1,3)], \quad [(1,2)]
\]

That is, there are three layers; the first layer has connections from wire 0 to wire 1 and from wire 2 to wire 3; the second a single connection from wire 1 to wire 3; and the third a single connection from wire 1 to wire 2.

type Layout = [((Size, Size))]

layout :: Circuit2 \rightarrow Layout

layout (Identity w) = [ ]

layout (Fan w) = [[(0, j) | j ← [1..w - 1]]]

layout (Above c d) = layout c ++ layout d

layout (Beside c d) = lwAlg wswAlg (width c) (layout c)

layout (Stretch w c) = map (map (connect ws)) (layout c)

shift w = map (pmap (w + !))

connect ws = pmap (pred c (scanl1 (+) ws)))

Here, \(pmap\) is the map function for homogeneous pairs:

\(pmap ((a \rightarrow b) \rightarrow (a, a) \rightarrow (b, b))\)

The function \(lwAlg\) is ‘long zip with’ \([13]\), which zips two lists together and returns a result as long as the longer argument. The binary operator is used to combine corresponding elements; if one list is shorter then the remaining elements of the other are simply copied.

\(lwAlg :: CircuitAlg \rightarrow CircuitAlg \rightarrow CircuitAlg\)

\(lwAlg (IdentityF w) = [([], w)]\)

\(lwAlg (FanF w) = [[(0, j) | j ← [1..w - 1]], w]\)

\(lwAlg (AboveF c d) = lwAlg (Above c d) (lwAlg w wAlg (width c) (lwAlg c d))\)

\(lwAlg (BesideF c d) = lwAlg (Beside c d) (lwAlg w wAlg (width c) (lwAlg c d))\)

\(lwAlg (StretchF ws c) = \text{map (map (connect ws)) (lwAlg c d)}\)

But even having achieved this, there is room for improvement. In the Beside and Stretch clauses, sublayouts are postprocessed using \(shift\) and \(map (map (connect ws))\) respectively. It would be more efficient to do this processing via an accumulating parameter \([3]\).
instead. In this case, a transformation on wire indices suffices as the accumulating parameter (‘\texttt{tlayout}’ stands for ‘transformed layout’):

\[
\texttt{tlayout} :: (\text{Size} \rightarrow \text{Size}) \rightarrow \text{Circuit}_2 \rightarrow \text{Layout} \\
\texttt{tlayout} f c = \text{map} (\text{map} \text{pmap} f) \ (\text{layout} c)
\]

Of course, \texttt{layout} = \texttt{tlayout id}, and it is a straightforward exercise to synthesize the following more efficient definition of \texttt{layout}:

\[
\texttt{layout} :: (\text{Size} \rightarrow \text{Size}) \rightarrow \text{Circuit}_2 \rightarrow \text{Layout} \\
\texttt{layout} f (\text{Identity} w) = [] \\
\texttt{layout} f (\text{Fan} w) = [(f 0 f j) | j \leftarrow [1..w-1]] \\
\texttt{layout} f (\text{Above} c d) = \text{layout} f c \downarrow \text{layout} f d \\
\texttt{layout} f (\text{Beside} c d) = \text{lwz}(++) (\text{layout} f c) \ (\text{layout} ((w+)of) d) \\
\]

\textbf{where} \texttt{w} = \text{width} \ c \\
\texttt{layout} f (\text{Stretch} w s c) = \text{layout} (\text{pred} \circ (\text{vs}!) \circ f) \ c \\
\textbf{where} \texttt{vs} = \text{scanl} I (+) \ w s

And how does this work out with a shallow embedding? Note that \texttt{layout} \ f \ is no longer a fold, because the accumulating parameter changes in some recursive calls. One might say that \texttt{layout} is a context-sensitive layout function, and the context may vary in recursive calls. But standard fold technology comes to the rescue once more: \texttt{layout} may not be a fold, but \texttt{flip layout} is—specifically, an accumulating fold.

\[
\texttt{tlwAlg} :: \text{CircuitAlg} ((\text{Size} \rightarrow \text{Size}) \rightarrow \text{Layout}, \text{Width}) \\
\texttt{tlwAlg} (\text{Identity} f w) = (f w \rightarrow [\text{w}]) \\
\texttt{tlwAlg} (\text{Fan} f w) = (f \rightarrow \lbrack (f 0 f j) | j \leftarrow [1..w-1] \rbrack, \text{w}) \\
\texttt{tlwAlg} (\text{Above} f c d) = (f \rightarrow \text{fst c} \circ f \circ \text{snd c}) \\
\texttt{tlwAlg} (\text{Beside} f c d) = (f \rightarrow \text{lwz}(++) (\text{fst c}) \ (\text{fst d} \circ \text{snd c}))
\]

\textbf{where} \texttt{vs} = \text{scanl} I (+) \ w s

The alert reader may have noted another source of inefficiency in \texttt{layout}, namely the uses of \texttt{++} and \texttt{lwz(++)} in the Above and Beside cases. These too can be removed, by introducing two more accumulating parameters, giving:

\[
\texttt{ulayout} :: (\text{Size} \rightarrow \text{Size}) \rightarrow \text{Layout} \rightarrow \text{Layout} \\
\texttt{ulayout} f l l' c = (\text{lwz}(++) (\text{map} (\text{map} \text{pmap} f) \ (\text{layout} c))) \ l \ l'
\]

(now ‘\texttt{ulayout}’ stands for ‘ultimate layout’). From this specification we can synthesize a definition that takes linear time in the ‘size’ of the circuit, for a reasonable definition of ‘size’. We leave the details as an exercise.

In fact, the standard interpretation \texttt{apply} given above is really another accumulating fold, in disguise. Rather than reading the type

\[
\texttt{apply} :: \text{Semigroup} a \Rightarrow \text{Circuit}_2 \rightarrow [a] \rightarrow [a]
\]

defining an interpretation of circuits as list transformers of type \texttt{Semigroup} \ a \Rightarrow ([a] \rightarrow [a]), one can read this as defining a context-dependent interpretation as an output list of type \texttt{Semigroup} \ a \Rightarrow [a], dependent on some input list of the same type. The interpretation is implemented in terms of an accumulating parameter; this is initially the input list, but it ‘accumulates’ by attrition via \texttt{splitAt} and \texttt{map last} \circ \texttt{bundle} \ w s as the evaluation proceeds.

4.4 Parametrized interpretations

We saw in Section 4.1 that it is not difficult to provide multiple interpretations with a shallow embedding, by constructing a tuple as the semantics of an expression and projecting the desired interpretation from the tuple. 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 [25].

But as we have also seen, all compositional interpretations conform to a common pattern: they are folds. So we can provide a shallow embedding as precisely that pattern—that is, in terms of a single parametrized interpretation, which is a higher-order value representing the fold.

\[
\texttt{newtype Circuit}_7 = C_7 \ {\text{unC}_7 :: \forall a. \text{CircuitAlg} a \rightarrow a} \\
\text{identity}_7 w = C_7 (\lambda h \rightarrow h \ (\text{Identity} f w)) \\
\text{fan}_7 w = C_7 (\lambda h \rightarrow h \ (\text{Fan} f w)) \\
\text{above}_7 x y = C_7 (\lambda h \rightarrow h \ (\text{Above} f (\text{unC}_7 x h) \ (\text{unC}_7 y h))) \\
\text{beside}_7 x y = C_7 (\lambda h \rightarrow h \ (\text{Beside} f (\text{unC}_7 x h) \ (\text{unC}_7 y h))) \\
\text{stretch}_7 w x = C_7 (\lambda h \rightarrow h \ (\text{Stretch} f \ (\text{unC}_7 x h)))
\]

(We need the \texttt{newtype} instead of a plain \texttt{type} synonym because of the quantified type.) This shallow encoding subsumes all others; it specializes to \texttt{depth} and \texttt{width}, and of course to any other fold:

\[
\text{width}_7 :: \text{Circuit}_7 \rightarrow \text{Width} \\
\text{width}_7 \circ \text{circuit} = \text{unC}_7 \circ \text{circuit} \circ \text{widthAlg} \\
\text{depth}_7 :: \text{Circuit}_7 \rightarrow \text{Depth} \\
\text{depth}_7 \circ \text{circuit} = \text{unC}_7 \circ \text{circuit} \circ \text{depthAlg}
\]

In fact, the shallow embedding provides a universal generic interpretation as the \textit{Church encoding} [15] of the AST—or more precisely, because it is typed, the \textit{Böhm–Berarducci encoding} [4].

Universality is witnessed by the observation that it is possible to recover the deep embedding from this one ‘mother of all shallow embeddings’ [8]:

\[
\texttt{deep} :: \text{Circuit}_7 \rightarrow \text{Circuit}_4 \\
\text{deep} \circ \text{circuit} = \text{unC}_7 \circ \text{circuit} \circ \text{In}
\]

(\text{So it turns out that the syntax of the DSL is not really as ephemeral in a shallow embedding as Boulton’s choice of terms [6] suggests.) And conversely, one can map from the deep embedding to the parametrized shallow embedding, and thence to any other shallow embedding:

\[
\text{shallow} :: \text{Circuit}_4 \rightarrow \text{Circuit}_7 \\
\text{shallow} = \text{foldC} \circ \text{shallowAlg} \\
\text{shallowAlg} :: \text{CircuitAlg}_7 \\
\text{shallowAlg} (\text{Identity} f w) = \text{identity}_7 w \\
\text{shallowAlg} (\text{Fan} f w) = \text{fan}_7 w \\
\text{shallowAlg} (\text{Above} f c d) = \text{above}_7 c d \\
\text{shallowAlg} (\text{Beside} f c d) = \text{beside}_7 c d \\
\text{shallowAlg} (\text{Stretch} f w s c) = \text{stretch}_7 w s c
\]

Moreover, \texttt{deep} and \texttt{shallow} are each other’s inverses, assuming parametricity [29].

4.5 Implicitly parametrized interpretations

The shallow embedding in Section 4.4 involves explicitly passing an algebra with which to interpret terms. That parameter may be passed implicitly instead, if it can be determined from the type of the interpretation. In Haskell, this can be done by defining a suitable type class:

\[
\texttt{class Circuit}_8 \circ \text{circuit where} \\
\text{identity}_8 :: \text{Size} \rightarrow \text{circuit} \\
\text{fan}_8 :: \text{Size} \rightarrow \text{circuit} \\
\text{above}_8 :: \text{circuit} \rightarrow \text{circuit} \rightarrow \text{circuit} \\
\text{beside}_8 :: \text{circuit} \rightarrow \text{circuit} \rightarrow \text{circuit} \\
\text{stretch}_8 :: \text{[Size]} \rightarrow \text{circuit} \rightarrow \text{circuit}
\]

To specify a particular interpretation, one defines an instance of the type class for the type of that interpretation. For example, here is the specification of the ‘width’ interpretation:

\[
\texttt{newtype Width}_8 = \text{Width} \ {\text{unWidth} :: \text{Int}}
\]
The *newtype* wrapper is often needed to allow multiple interpretations over the same underlying type; for example, we can provide both 'width' and 'depth' interpretations over integers:

```haskell
newtype Depth8 = Depth { unDepth :: Int }
```

The conventional implementation of type classes [31] involves constructing a *dictionary* for each type in the type class, and generating code that selects and passes the appropriate dictionary as an additional parameter to each overloaded member function (identity8, fan8 etc). For an instance of the type class, the dictionary type is equivalent to `CircuitAlg c`. Indeed, we might have defined instead:

```haskell
class Circuit8 c where
  alg :: CircuitAlg c
```

so that the dictionary type is literally a `CircuitAlg c`; the Böhm–Berarducci and type-class approaches are really very similar.

### 4.6 Intermediate interpretations

Good practice in the design of embedded DSLs is to distinguish between a minimal ‘core’ language and a more useful ‘everyday’ language [27]. The former is more convenient for the language designer, but the latter more convenient for the language user. This apparent tension can be resolved by defining the additional constructs that correspond to core constructs directly; the derived constructs are defined by translation.

```haskell
instance Circuit8 Width8 where
  identity8 w = Width w
  fan8 w = Width w
  above8 x y = x
  beside8 x y = Width (unWidth x + unWidth y)
  stretch8 ws x = x

Some of the wrapping and unwrapping of `Width8` and `Depth8` values could be avoided by installing these types as instances of the `Num` and `Ord` type classes; this can even be done automatically in GHC, by exploiting the ‘Generalized Newtype Deriving’ extension.

One might see this as a shallow embedding, with the carrier `CoreCircuit` itself the deep embedding of a different, smaller language; the core constructs are implemented directly as constructors of `CoreCircuit`, and non-core constructs as a kind of ‘smart constructor’.

This suggests that ‘deep’ and ‘shallow’ do not form a dichotomy, but rather are two extreme points on a scale of embedding depth. Augustsson [2] discusses representations of intermediate depth, in which some constructs have deep embeddings and some shallow. In particular, for a language with a ‘semantics’ in the form of generated assembly code, the deeply embedded constructs will persist as generated code, whereas those with shallow embeddings will get translated away at ‘compile time’. Augustsson calls these *neritic* embeddings, after the region of the sea between the shore and the edge of the continental shelf.

### 4.7 Modular interpretations

The previous section explored cutting down the grammar of circuits by eliminating a constructor. Conversely, one might extend the grammar by adding constructors. Indeed, in addition to the ‘left stretch’ combinator we have used, Hinze [14] also provides a ‘right stretch’ combinator, which connects the first rather than the last wire of each bundle to the inner circuit. This is not needed in the core language, because it can be built out of existing components:

```haskell
rstretch (ws ++ [w + 1]) c = stretch (1 : ws) c 'beside' identity w
```

So one might extend the grammar of the everyday language, as embodied in the functor `CircuitF` or the type class `Circuit8`, to incorporate this additional operator, but still use `CoreCircuit` as the actual representation.

Alternatively, one might hope for a modular technique for assembling embedded languages and their interpretations from parts, so that it is straightforward to add additional constructors like ‘right stretch’. Swierstra’s *datatypes à la carte* machinery [28] provides precisely such a thing, going some way towards addressing the expression problem discussed in Section 2.

The key idea is to represent each constructor separately:

```haskell
data Identity11 c = Identity11 Size deriving Functor
data Fan11 c = Fan11 Size deriving Functor
data Above11 c = Above11 c c deriving Functor
data Beside11 c = Beside11 c c deriving Functor
data Stretch11 c = Stretch11 [Size] c deriving Functor
```

with a right-associating ‘sum’ operator for combining them:

```haskell
data (f :+: g) e = Inl (f e) | Inr (g e) deriving Functor
```

One can assemble a functor from these components and make a deep embedding from it. For example, the sum of functors `CircuitF11` is equivalent to `CircuitF` from the start of Section 4, and its fixpoint `Circuit11` to `Circuit4`:

```haskell

data Fixf = In (f (Fixf))
type Circuit11 = Fix CircuitF11
```

This works, but it is rather clumsy. In particular, an expression of type `Circuit11` involves a mess of `Inl`, `Inr` and `In` constructors, as seen in this rendition of the circuit in Figure 4:
Then we provide instances for each of the relevant constructors. For example, this width function works for the flexibly typed circuit `stretchfan` above:

```
instance WidthAlg Fan where
  widthAlg (Fan w) = w
instance WidthAlg Stretch where
  widthAlg (Stretch ws x) = sum ws
```

For example, the circuit does not need to be given a specific type first:

```
> width11 (stretchfan :: Circuit11)
8
```

These algebra fragments together constitute the essence of an implicitly parametrized shallow embedding. But the main benefit of the à la carte approach is that it is easy to add new constructors. We just need to add the datatype constructor as a functor, and provide a smart constructor:

```
data RStretch11 c = RStretch11 [Size] c deriving Functor
```

Now the circuit in Figure 4 can be expressed using right stretch instead of left stretch:

```
rstretchfan :: (Identity11 ::< f, Fan11 ::< f, Beside11 ::< f, RStretch11 ::< f) ⇒ [Width] → Fix f → Fix f
rstretchfan = beside11 (Identity11 2) (rstretch11 1 [2,3,1] (fan11 3))
```

When adding new constructors such as `RStretch11`, it is tempting to provide an instance for each of the interpretations of interest, such as `WidthAlg`. However, this is an unnecessary duplication of effort when `rstretch11` can itself be simulated out of existing components. We might instead write a function that `handles` the `RStretch11` constructor:

```
handle :: (Stretch11 ::< f, Beside11 ::< f, Identity11 ::< f) ⇒ Fix (RStretch11 ::< f) → Fix f
handle (In (Inr (RStretch11 ws c))) = stretch11 (1 :: ws') (handle c) 'beside11' 'identity11 w
where (ws',w) = (init ws, last ws - 1)
handle (In (Inr other)) = In (fmap handle other)
```

Here, we recursively translate all instances of `RStretch11` into other constructors. This technique is at the heart of the effects and handlers approach [22], although the setting there uses the free monad rather than `Fix`. With this in place, we can first handle all of the `RStretch11` constructors before passing the result on to an interpretation function such as `width11` that need not deal with `RStretch11`s. This method of interpreting only a core fragment of syntax might not be optimally efficient, but of course we still leave open the possibility of providing a specialized instance if that is an issue.

5. **Discussion**

The essential observation made here—that shallow embeddings correspond to the algebras of folds over the abstract syntax captured by a deep embedding—is surely not new. For example, it was probably known to Reynolds [26], who contrasted deep embeddings (‘user defined types’) and shallow (‘procedural data structures’), and observed that the former were free algebras; but he didn’t explicitly discuss anything corresponding to folds.

It is also implicit in the finally tagless approach [8], which uses a shallow embedding and observes that ‘this representation makes it trivial to implement a primitive recursive function over
object terms', providing an interface that such functions should implement; but this comment is made rather in passing, and their focus is mainly on staging and partial evaluation. The observation is more explicit in Kiselyov’s lecture notes on the finally tagless approach [21], which go into more detail on compositionality; he makes the connection to “denotational semantics, which is required to be compositional”, and observes that “making context explicit turns seemingly non-compositional operations compositional”. The finally tagless approach also covers DSLs with binding constructs, which we have ignored here.

Neither is it a new observation that algebraic datatypes (such as 
\texttt{Circuit}_4\) and their Böhm–Berarducci encodings (such as 
\texttt{Circuit}_7\) are equivalent. And of course, none of this is specific in any way to the 
\texttt{Circuit} DSL; a datatype-generic version of the story can be told, by abstracting away from the shape functor \texttt{Circuit}_F—the reader may enjoy working out the details.

Nevertheless, the observation that shallow embeddings corre-
spond to the algebras of folds over deep embeddings seems not to be widely appreciated; at least, we have been unable to find an explicit statement to this effect, either in the DSL literature or elsewhere. And it makes a nice application of folds: many results about folds evidently have interesting statements about shallow embeddings as corecurrs. The three generalizations of folds (banana split, mutu-
morphisms, and accumulating parameters) exploited in Section 4 are all special cases of \textit{adjoint fold} [16, 17]; perhaps other adjoint folds yield other interesting insights about shallow embeddings?

Acknowledgements

This paper arose from ideas discussed at the Summer School on Domain Specific Languages in Cluj-Napoca in July 2013 [12]; JG thanks the organizers for the invitation to lecture there. José Pedro Magalhães, Ralf Hinze, Jacques Carette, James McK
innna, and the anonymous reviewers all made helpful comments, and Oleg Kiselyov gave many constructive criticisms and much inspira-
tion, for which we are very grateful. This work has been funded by EPSRC grant number EP/J010995/1, on Unifying Theories of Generic Programming.

References

[12] Jeremy Gibbons. Functional programming for domain-specific lan-
[18] Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. Formal derivation of efficient parallel programs by construction of list homomor-