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:

$$
\begin{array}{l}
\textbf{type}\ \mathit{Expr}_1 = \ldots \\
\mathit{lit}\ \ :: \mathit{Integer} \qquad\qquad \to \mathit{Expr}_1 \\
\mathit{add} :: \mathit{Expr}_1 \to \mathit{Expr}_1 \to \mathit{Expr}_1
\end{array}
$$

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

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

$$
\begin{array}{l}
\textbf{data}\ \mathit{Expr}_2 :: * \ \textbf{where} \\
\quad \mathit{Lit}\ \ :: \mathit{Integer} \qquad\qquad \to \mathit{Expr}_2 \\
\quad \mathit{Add} :: \mathit{Expr}_2 \to \mathit{Expr}_2 \to \mathit{Expr}_2 \\
\mathit{lit}\ n \quad\ = \mathit{Lit}\ n \\
\mathit{add}\ x\ y = \mathit{Add}\ x\ y
\end{array}
$$

(We have used Haskell's 'generalized algebraic datatype' notation, in order to make the types of the constructors *Lit* and *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:

ICFP '14, September 1–6, 2014, Gothenburg, Sweden.

http://dx.doi.org/10.1145/2628136.2628138

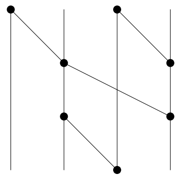

Figure 1. The Brent–Kung parallel prefix circuit of width 4

$eval_2 :: Expr_2 \to Integer$
$eval_2\ (Lit\ n) \quad = n$
$eval_2\ (Add\ x\ y) = eval_2\ x + eval_2\ y$

This might be used as follows:

$>\ eval_2\ (Add\ (Add\ (Lit\ 3)\ (Lit\ 4))\ (Lit\ 5))$
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:

type $Expr_3 = Integer$

$lit\ n \quad = n$
$add\ x\ y = x + y$

$eval_3 :: Expr_3 \to Integer$
$eval_3\ n = n$

This might be used as follows:

$>\ eval_3\ (add\ (add\ (lit\ 3)\ (lit\ 4))\ (lit\ 5))$
12

We have used subscripts to distinguish different representations of morally 'the same' functions ($eval_2$ and $eval_3$) and types ($Expr_2$ and $Expr_3$). 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 *Lit* and *Add* in the deep embedding do none of the work, leaving this entirely to the observation function *eval*; in contrast, in the shallow embedding, the language constructs *lit* and *add* do all the work, and the observer $eval_3$ 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 *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 *lit* and *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 *lit* and *add* in the shallow embedding coincide with those of *Lit* and *Add* in the deep embedding; moreover, the definitions of *lit* and *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.

Figure 2. Identity circuit *identity* 4 and fan circuit *fan* 4 of width 4

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 $\bullet$, a prefix computation of width $n > 0$ takes a sequence $x_1, x_2, ..., x_n$ of inputs and produces the sequence $x_1, x_1 \bullet x_2, ..., x_1 \bullet x_2 \bullet \cdots \bullet x_n$ 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 $\bullet$, 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, x_4 to $x_1, x_1 \bullet x_2, x_1 \bullet x_2 \bullet x_3, x_1 \bullet x_2 \bullet x_3 \bullet x_4$.

Such circuits can be constructed using the following operators:

type $Size = Int$ -- positive
type $Circuit_1 = ...$

$identity :: Size \to Circuit_1$
$fan \quad :: Size \to Circuit_1$
$above \quad :: Circuit_1 \to Circuit_1 \to Circuit_1$
$beside \quad :: Circuit_1 \to Circuit_1 \to Circuit_1$
$stretch \quad :: [Size] \to Circuit_1 \to Circuit_1$

The most basic building block is the identity circuit, *identity n*, which creates a circuit consisting of *n* parallel wires that copy input to output. The other primitive is the fan circuit; *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 *identity* and *fan* of width 4 are shown in Figure 2.

Then there are three combinators for circuits. The series or vertical composition, *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, *beside 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, *stretch ws c*, takes a non-empty list of positive widths $ws = [w_1, ..., w_n]$ of length *n*, and a circuit *c* of width *n*, and "stretches" *c* out to width *sum ws* 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

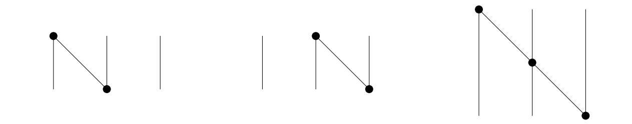

Figure 3. The construction of a parallel prefix circuit of width 3

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

```
(fan 2 `beside` fan 2) `above`
stretch [2,2] (fan 2) `above`
(identity 1 `beside` fan 2 `beside` 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 :: Size → Circuit₁
brentkung 1 = identity 1
brentkung w
  = (row (replicate u (fan 2)) `pad` w) `above`
    (stretch (replicate u 2) (brentkung u) `pad` w) `above`
    (row (identity 1 : replicate v (fan 2)) `pad` (w − 1))
  where (u, v)      = (w `div` 2, (w − 1) `div` 2)
        c `pad` w = if even w then c else c `beside` identity 1
        row         = foldr1 beside
```

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

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

```
data Circuit₂ :: ∗ where
  Identity :: Size →                     Circuit₂
  Fan      :: Size →                     Circuit₂
  Above    :: Circuit₂ → Circuit₂ → Circuit₂
  Beside   :: Circuit₂ → Circuit₂ → Circuit₂
  Stretch  :: [Size] → Circuit₂ →     Circuit₂
```

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

```
type Width = Int

width₂ :: Circuit₂ → Width
width₂ (Identity w)   = w
width₂ (Fan w)        = w
width₂ (Above x y)    = width₂ x
width₂ (Beside x y)   = width₂ x + width₂ y
width₂ (Stretch ws x) = sum ws
```

Note that $width_2$ is compositional: it is a fold over the abstract syntax of $Circuit_2$s. That makes it a suitable semantics for a shallow embedding. That is, we could represent circuits directly by their widths, as follows:

```
type Circuit₃ = Width

identity w   = w
```

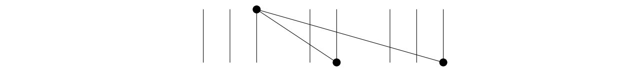

Figure 4. A 3-fan stretched out by widths $[3,2,3]$

```
fan w          = w
above x y      = x
beside x y     = x + y
stretch ws x = sum ws

width₃ :: Circuit₃ → Width
width₃ = id
```

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 $Circuit_2$ above has five constructors; *Identity* and *Fan* each take one argument of the fixed type *Size*; *Above* and *Beside* take two arguments, both recursive occurrences; *Stretch* takes two arguments, one of which is the fixed type $[Size]$, 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 *Functor* type class:

```
class Functor f where
  fmap :: (a → b) → (f a → f b)
```

For $Circuit_2$, the shape is given by *CircuitF* as follows, where the parameter x marks the recursive spots:

```
data CircuitF :: ∗ → ∗ where
  IdentityF :: Size →          CircuitF x
  FanF      :: Size →          CircuitF x
  AboveF    :: x → x →         CircuitF x
  BesideF   :: x → x →         CircuitF x
  StretchF  :: [Size] → x → CircuitF x
instance Functor CircuitF where
  fmap f (IdentityF w)    = IdentityF w
  fmap f (FanF w)         = FanF w
  fmap f (AboveF x₁ x₂)  = AboveF (f x₁) (f x₂)
  fmap f (BesideF x₁ x₂) = BesideF (f x₁) (f x₂)
  fmap f (StretchF ws x)  = StretchF ws (f x)
```

We can use this shape functor as the basis of an alternative definition of the algebraic datatype $Circuit_2$:

```
data Circuit₄ = In (CircuitF Circuit₄)
```

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

```
type CircuitAlg a = CircuitF a → a
```

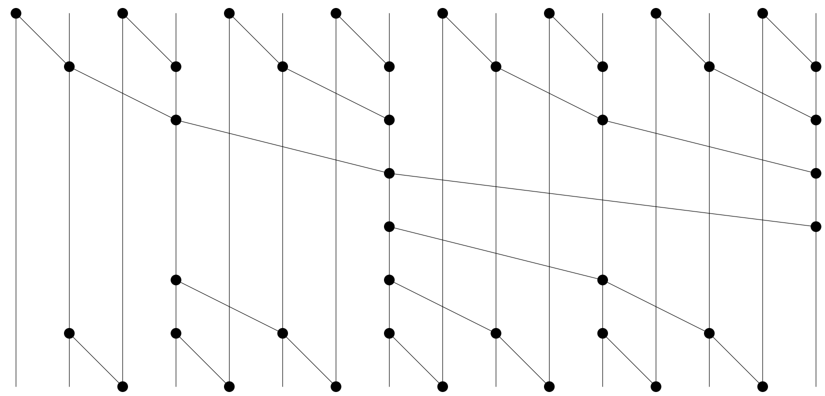

Figure 5. The Brent–Kung parallel prefix circuit of width 16

Such an algebra is precisely the information needed to fold a data structure:

$$
\begin{array}{l}
\mathit{foldC} :: \mathit{CircuitAlg}\ a \to \mathit{Circuit}_4 \to a \\
\mathit{foldC}\ h\ (\mathit{In}\ x) = h\ (\mathit{fmap}\ (\mathit{foldC}\ h)\ x)
\end{array}
$$

For example, *width* is a fold for the deeply embedded DSL of shape *CircuitF*, and is determined by the following algebra:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{widthAlg} :: \mathit{CircuitAlg}\ \mathit{Width}} \\
\mathit{widthAlg}\ (\mathit{IdentityF}\ w) & = w \\
\mathit{widthAlg}\ (\mathit{FanF}\ w) & = w \\
\mathit{widthAlg}\ (\mathit{AboveF}\ x\ y) & = x \\
\mathit{widthAlg}\ (\mathit{BesideF}\ x\ y) & = x + y \\
\mathit{widthAlg}\ (\mathit{StretchF}\ ws\ x) & = \mathit{sum}\ ws
\end{array}
$$

$$
\begin{array}{l}
\mathit{width}_4 :: \mathit{Circuit}_4 \to \mathit{Width} \\
\mathit{width}_4 = \mathit{foldC}\ \mathit{widthAlg}
\end{array}
$$

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.

4.1 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.

$$\textbf{type}\ \mathit{Depth} = \mathit{Int}$$

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{depthAlg} :: \mathit{CircuitAlg}\ \mathit{Depth}} \\
\mathit{depthAlg}\ (\mathit{IdentityF}\ w) & = 0 \\
\mathit{depthAlg}\ (\mathit{FanF}\ w) & = 1 \\
\mathit{depthAlg}\ (\mathit{AboveF}\ x\ y) & = x + y \\
\mathit{depthAlg}\ (\mathit{BesideF}\ x\ y) & = x\ `\mathit{max}`\ y \\
\mathit{depthAlg}\ (\mathit{StretchF}\ ws\ x) & = x
\end{array}
$$

$$
\begin{array}{l}
\mathit{depth}_4 :: \mathit{Circuit}_4 \to \mathit{Depth} \\
\mathit{depth}_4 = \mathit{foldC}\ \mathit{depthAlg}
\end{array}
$$

But what about with a shallow embedding? With this approach, circuits can only have a single semantics, so how do we accommodate finding 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.

$$\textbf{type}\ \mathit{Circuit}_5 = (\mathit{Width}, \mathit{Depth})$$

Now the observation functions width_5 and depth_5 become projections, rather than just the identity function.

$$
\begin{array}{l}
\mathit{width}_5 :: \mathit{Circuit}_5 \to \mathit{Width} \\
\mathit{width}_5 = \mathit{fst}
\end{array}
$$

$$
\begin{array}{l}
\mathit{depth}_5 :: \mathit{Circuit}_5 \to \mathit{Depth} \\
\mathit{depth}_5 = \mathit{snd}
\end{array}
$$

The individual operations can be defined much as before, just by projecting the relevant components out of the pair:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{wdAlg} :: \mathit{CircuitAlg}\ \mathit{Circuit}_5} \\
\mathit{wdAlg}\ (\mathit{IdentityF}\ w) & = (w, 0) \\
\mathit{wdAlg}\ (\mathit{FanF}\ w) & = (w, 1) \\
\mathit{wdAlg}\ (\mathit{AboveF}\ x\ y) & = (\mathit{width}_5\ x, \mathit{depth}_5\ x + \mathit{depth}_5\ y) \\
\mathit{wdAlg}\ (\mathit{BesideF}\ x\ y) & = (\mathit{width}_5\ x + \mathit{width}_5\ y, \\
 & \quad\ \mathit{depth}_5\ x\ `\mathit{max}`\ \mathit{depth}_5\ y) \\
\mathit{wdAlg}\ (\mathit{StretchF}\ ws\ x) & = (\mathit{sum}\ ws, \mathit{depth}_5\ x)
\end{array}
$$

This algebra is the essence of the shallow embedding; for example,

$$\mathit{identity}_5\ w = \mathit{wdAlg}\ (\mathit{IdentityF}\ w)$$

and so on. Of course, this works better under lazy than under eager evaluation: if only one of the two interpretations of an expression is needed, only that one is evaluated. And it's rather clumsy from a modularity perspective; we will return to this point later.

Seen from the fold perspective, this step is no surprise: the 'banana split law' [10] tells us that tupling two independent folds gives another fold, so multiple interpretations can be provided in the shallow embedding nearly as easily as in the deep embedding.

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, 'compositionality' 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*:

$$\textbf{type}\ \mathit{WellSized} = \mathit{Bool}$$

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{wellSized} :: \mathit{Circuit}_2 \to \mathit{WellSized}} \\
\mathit{wellSized}\ (\mathit{Identity}\ w) & = \mathit{True} \\
\mathit{wellSized}\ (\mathit{Fan}\ w) & = \mathit{True}
\end{array}
$$

$$
\begin{aligned}
&\mathit{wellSized}\ (\mathit{Above}\ x\ y) &&= \mathit{wellSized}\ x \wedge \mathit{wellSized}\ y \\
& &&\quad \wedge \mathit{width}\ x \equiv \mathit{width}\ y \\
&\mathit{wellSized}\ (\mathit{Beside}\ x\ y) &&= \mathit{wellSized}\ x \wedge \mathit{wellSized}\ y \\
&\mathit{wellSized}\ (\mathit{Stretch}\ \mathit{ws}\ x) &&= \mathit{wellSized}\ x \wedge \mathit{length}\ \mathit{ws} \equiv \mathit{width}\ x
\end{aligned}
$$

This is a non-compositional interpretation of the abstract syntax, because *wellSized* sometimes depends on the *width* of subcircuits as well as their recursive image under *wellSized*. In other words, *wellSized* is not a fold, and there is no corresponding *CircuitAlg*.

What can we do about such non-compositional interpretations in the shallow embedding? Again, fold theory comes to the rescue: *wellSized* and *width* together form a *mutumorphism* [10]—that is, two mutually dependent folds—and the tuple of these two functions again forms a fold. (In fact, this is a special case, a *zygomorphism* [10], since the dependency is only one-way. Simpler still, we have seen another special case in the banana split above, where neither of the two folds depends on the other.)

$$
\begin{aligned}
&\mathbf{type}\ \mathit{Circuit}_6 = (\mathit{WellSized}, \mathit{Width}) \\
&\mathit{wswAlg} :: \mathit{CircuitAlg}\ \mathit{Circuit}_6 \\
&\mathit{wswAlg}\ (\mathit{IdentityF}\ w) &&= (\mathit{True}, w) \\
&\mathit{wswAlg}\ (\mathit{FanF}\ w) &&= (\mathit{True}, w) \\
&\mathit{wswAlg}\ (\mathit{AboveF}\ x\ y) &&= (\mathit{fst}\ x \wedge \mathit{fst}\ y \wedge \mathit{snd}\ x \equiv \mathit{snd}\ y, \mathit{snd}\ x) \\
&\mathit{wswAlg}\ (\mathit{BesideF}\ x\ y) &&= (\mathit{fst}\ x \wedge \mathit{fst}\ y, \mathit{snd}\ x + \mathit{snd}\ y) \\
&\mathit{wswAlg}\ (\mathit{StretchF}\ \mathit{ws}\ x) &&= (\mathit{fst}\ x \wedge \mathit{length}\ \mathit{ws} \equiv \mathit{snd}\ x, \mathit{sum}\ \mathit{ws})
\end{aligned}
$$

So although $\mathit{wellSized} = \mathit{fst} \circ \mathit{foldC}\ \mathit{wswAlg}$ is not a fold, it is manifestly clear that *foldC wswAlg* is. Tupling functions in this way is analogous to strengthening the invariant of an imperative loop to record additional information [20], and is a standard trick in program calculation [18].

Another example of a dependent interpretation is provided by what Hinze [14] calls the *standard model* of the circuit: its interpretation as a computation. As discussed in the introduction, this is defined in terms of an associative binary operator ($\bullet$), which we capture by the following type class:

$$
\begin{aligned}
&\mathbf{class}\ \mathit{Semigroup}\ s\ \mathbf{where} \\
&\quad (\bullet) :: s \to s \to s \quad \text{-- } \bullet \text{ is associative}
\end{aligned}
$$

The interpretation *apply* interprets a circuit of width n as a function operating on lists of length n:

$$
\begin{aligned}
&\mathit{apply} :: \mathit{Semigroup}\ a \Rightarrow \mathit{Circuit}_2 \to [a] \to [a] \\
&\mathit{apply}\ (\mathit{Identity}\ w)\ \ \mathit{xs} &&= \mathit{xs} \\
&\mathit{apply}\ (\mathit{Fan}\ w)\ \ (x : \mathit{xs}) &&= x : \mathit{map}\ (x\bullet)\ \mathit{xs} \\
&\mathit{apply}\ (\mathit{Above}\ c\ d)\ \ \mathit{xs} &&= \mathit{apply}\ d\ (\mathit{apply}\ c\ \mathit{xs}) \\
&\mathit{apply}\ (\mathit{Beside}\ c\ d)\ \ \mathit{xs} &&= \mathit{apply}\ c\ \mathit{ys} \mathbin{+\!\!+} \mathit{apply}\ d\ \mathit{zs} \\
&\quad \mathbf{where}\ (\mathit{ys}, \mathit{zs}) = \mathit{splitAt}\ (\mathit{width}\ c)\ \mathit{xs} \\
&\mathit{apply}\ (\mathit{Stretch}\ \mathit{ws}\ c)\ \ \mathit{xs} &&= \mathit{concat} \\
&\quad (\mathit{zipWith}\ \mathit{snoc}\ (\mathit{map}\ \mathit{init}\ \mathit{xss})\ (\mathit{apply}\ c\ (\mathit{map}\ \mathit{last}\ \mathit{xss}))) \\
&\quad \mathbf{where}\ \mathit{xss} = \mathit{bundle}\ \mathit{ws}\ \mathit{xs}
\end{aligned}
$$

Here, *snoc* is ‘cons’ backwards,

$$\mathit{snoc}\ \mathit{ys}\ z = \mathit{ys} \mathbin{+\!\!+} [z]$$

and *bundle ws xs* groups the list *xs* into bundles of widths *ws*, assuming that $\mathit{sum}\ \mathit{ws} \equiv \mathit{length}\ \mathit{xs}$:

$$
\begin{aligned}
&\mathit{bundle} :: \mathit{Integral}\ i \Rightarrow [i] \to [a] \to [[a]] \\
&\mathit{bundle}\ []\ \ [] = [] \\
&\mathit{bundle}\ (w : \mathit{ws})\ \mathit{xs} = \mathit{ys} : \mathit{bundle}\ \mathit{ws}\ \mathit{zs} \\
&\quad \mathbf{where}\ (\mathit{ys}, \mathit{zs}) = \mathit{splitAt}\ w\ \mathit{xs}
\end{aligned}
$$

The *apply* interpretation is another zygomorphism, because in the *Beside* case *apply* depends on *width c* as well as *apply c* and *apply d*. And indeed, Hinze’s ‘standard model’ [14] comprises both the list transformer and the width, tupled together.

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:

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

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.

$$
\begin{aligned}
&\mathbf{type}\ \mathit{Layout} = [[(\mathit{Size}, \mathit{Size})]] \\
&\mathit{layout} :: \mathit{Circuit}_2 \to \mathit{Layout} \\
&\mathit{layout}\ (\mathit{Identity}\ w) &&= [] \\
&\mathit{layout}\ (\mathit{Fan}\ w) &&= [[(0,j) \mid j \leftarrow [1\,..\,w-1]]] \\
&\mathit{layout}\ (\mathit{Above}\ c\ d) &&= \mathit{layout}\ c \mathbin{+\!\!+} \mathit{layout}\ d \\
&\mathit{layout}\ (\mathit{Beside}\ c\ d) &&= \mathit{lzw}\ (\mathbin{+\!\!+})\ (\mathit{layout}\ c) \\
& &&\qquad (\mathit{shift}\ (\mathit{width}\ c)\ (\mathit{layout}\ d)) \\
&\mathit{layout}\ (\mathit{Stretch}\ \mathit{ws}\ c) &&= \mathit{map}\ (\mathit{map}\ (\mathit{connect}\ \mathit{ws}))\ (\mathit{layout}\ c) \\
&\mathit{shift}\ w &&= \mathit{map}\ (\mathit{map}\ (\mathit{pmap}\ (w+))) \\
&\mathit{connect}\ \mathit{ws} &&= \mathit{pmap}\ (\mathit{pred} \circ ((\mathit{scanl1}\ (+)\ \mathit{ws})!!))
\end{aligned}
$$

Here, *pmap* is the map function for homogeneous pairs:

$$
\begin{aligned}
&\mathit{pmap} :: (a \to b) \to (a,a) \to (b,b) \\
&\mathit{pmap}\ f\ (x,y) = (f\ x, f\ y)
\end{aligned}
$$

The function *lzw* 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.

$$
\begin{aligned}
&\mathit{lzw} :: (a \to a \to a) \to [a] \to [a] \to [a] \\
&\mathit{lzw}\ f\ []\ \ \mathit{ys} &&= \mathit{ys} \\
&\mathit{lzw}\ f\ \mathit{xs}\ \ [] &&= \mathit{xs} \\
&\mathit{lzw}\ f\ (x : \mathit{xs})\ (y : \mathit{ys}) &&= f\ x\ y : \mathit{lzw}\ f\ \mathit{xs}\ \mathit{ys}
\end{aligned}
$$

The *layout* interpretation is yet another zygomorphism, because *layout* (*Beside c d*) depends on *width c* as well as *layout c* and *layout d*. In fact, in general we need the width of the circuit anyway in order to determine the layout, in case the rightmost wire is not connected to the others. So the techniques discussed above will allow us to express the layout as a shallow embedding, whose essence is as follows:

$$
\begin{aligned}
&\mathit{lwAlg} :: \mathit{CircuitAlg}\ (\mathit{Layout}, \mathit{Width}) \\
&\mathit{lwAlg}\ (\mathit{IdentityF}\ w) &&= ([], w) \\
&\mathit{lwAlg}\ (\mathit{FanF}\ w) &&= ([[(0,j) \mid j \leftarrow [1\,..\,w-1]]], w) \\
&\mathit{lwAlg}\ (\mathit{AboveF}\ c\ d) &&= (l_1 \mathbin{+\!\!+} l_2, w_2) \\
&\quad \mathbf{where}\ (l_1, w_1) = c; (l_2, w_2) = d \\
&\mathit{lwAlg}\ (\mathit{BesideF}\ c\ d) &&= (\mathit{lzw}\ (\mathbin{+\!\!+})\ l_1\ (\mathit{shift}\ w_1\ l_2), w_1 + w_2) \\
&\quad \mathbf{where}\ (l_1, w_1) = c; (l_2, w_2) = d \\
&\mathit{lwAlg}\ (\mathit{StretchF}\ \mathit{ws}\ (l, w)) &&= (\mathit{map}\ (\mathit{map}\ (\mathit{connect}\ \mathit{ws}))\ l, \mathit{sum}\ \mathit{ws})
\end{aligned}
$$

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 ('*tlayout*' stands for 'transformed *layout*'):

$$
\begin{aligned}
&\mathit{tlayout} :: (\mathit{Size} \to \mathit{Size}) \to \mathit{Circuit}_2 \to \mathit{Layout}\\
&\mathit{tlayout}\ f\ c = \mathit{map}\ (\mathit{map}\ (\mathit{pmap}\ f))\ (\mathit{layout}\ c)
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathit{tlayout} :: (\mathit{Size} \to \mathit{Size}) \to \mathit{Circuit}_2 \to \mathit{Layout}\\
&\mathit{tlayout}\ f\ (\mathit{Identity}\ w) &&= [\,]\\
&\mathit{tlayout}\ f\ (\mathit{Fan}\ w) &&= [[(f\ 0, f\ j) \mid j \leftarrow [1\,..\,w-1]]]\\
&\mathit{tlayout}\ f\ (\mathit{Above}\ c\ d) &&= \mathit{tlayout}\ f\ c +\!\!+ \mathit{tlayout}\ f\ d\\
&\mathit{tlayout}\ f\ (\mathit{Beside}\ c\ d) &&= \mathit{lzw}\ (+\!\!+)\ (\mathit{tlayout}\ f\ c)\\
& &&\qquad\qquad\quad (\mathit{tlayout}\ ((w+) \circ f)\ d)\\
&\quad \mathbf{where}\ w = \mathit{width}\ c\\
&\mathit{tlayout}\ f\ (\mathit{Stretch}\ \mathit{ws}\ c) = \mathit{tlayout}\ (\mathit{pred} \circ (\mathit{vs}!!) \circ f)\ c\\
&\quad \mathbf{where}\ \mathit{vs} = \mathit{scanl1}\ (+)\ \mathit{ws}
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathit{tlwAlg} :: \mathit{CircuitAlg}\ ((\mathit{Size} \to \mathit{Size}) \to \mathit{Layout}, \mathit{Width})\\
&\mathit{tlwAlg}\ (\mathit{IdentityF}\ w) &&= (\lambda f \to [\,], w)\\
&\mathit{tlwAlg}\ (\mathit{FanF}\ w) &&= (\lambda f \to [[(f\ 0, f\ j) \mid j \leftarrow [1\,..\,w-1]]], w)\\
&\mathit{tlwAlg}\ (\mathit{AboveF}\ c\ d) &&= (\lambda f \to \mathit{fst}\ c\ f +\!\!+ \mathit{fst}\ d\ f, \mathit{snd}\ c)\\
&\mathit{tlwAlg}\ (\mathit{BesideF}\ c\ d) &&= (\lambda f \to \mathit{lzw}\ (+\!\!+)\ (\mathit{fst}\ c\ f)\\
& &&\qquad\qquad\qquad (\mathit{fst}\ d\ ((\mathit{snd}\ c+) \circ f)),\\
& &&\quad \mathit{snd}\ c + \mathit{snd}\ d)\\
&\mathit{tlwAlg}\ (\mathit{StretchF}\ \mathit{ws}\ c) = (\lambda f \to \mathit{fst}\ c\ (\mathit{pred} \circ (\mathit{vs}!!) \circ f), \mathit{sum}\ \mathit{ws})\\
&\quad \mathbf{where}\ \mathit{vs} = \mathit{scanl1}\ (+)\ \mathit{ws}
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathit{ulayout} :: (\mathit{Size} \to \mathit{Size}) \to \mathit{Layout} \to \mathit{Layout} \to\\
&\qquad\qquad \mathit{Circuit}_2 \to \mathit{Layout}\\
&\mathit{ulayout}\ f\ l\ l'\ c = (\mathit{lzw}\ (+\!\!+)\ (\mathit{map}\ (\mathit{map}\ (\mathit{pmap}\ f))\ (\mathit{layout}\ c))\ l) +\!\!+ l'
\end{aligned}
$$

(now '*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 *apply* given above is really another accumulating fold, in disguise. Rather than reading the type

$$\mathit{apply} :: \mathit{Semigroup}\ a \Rightarrow \mathit{Circuit}_2 \to [a] \to [a]$$

as defining an interpretation of circuits as list transformers of type $\mathit{Semigroup}\ a \Rightarrow ([a] \to [a])$, one can read it as defining a context-dependent interpretation as an output list of type $\mathit{Semigroup}\ a \Rightarrow [a]$, 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 *splitAt* and $\mathit{map}\ \mathit{last} \circ \mathit{bundle}\ \mathit{ws}$ 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.

$$
\begin{aligned}
&\mathbf{newtype}\ \mathit{Circuit}_7 = C_7\ \{\mathit{unC}_7 :: \forall a\,.\ \mathit{CircuitAlg}\ a \to a\}\\
&\mathit{identity}_7\ w &&= C_7\ (\lambda h \to h\ (\mathit{IdentityF}\ w))\\
&\mathit{fan}_7\ w &&= C_7\ (\lambda h \to h\ (\mathit{FanF}\ w))\\
&\mathit{above}_7\ x\ y &&= C_7\ (\lambda h \to h\ (\mathit{AboveF}\ (\mathit{unC}_7\ x\ h)\ (\mathit{unC}_7\ y\ h)))\\
&\mathit{beside}_7\ x\ y &&= C_7\ (\lambda h \to h\ (\mathit{BesideF}\ (\mathit{unC}_7\ x\ h)\ (\mathit{unC}_7\ y\ h)))\\
&\mathit{stretch}_7\ \mathit{ws}\ x &&= C_7\ (\lambda h \to h\ (\mathit{StretchF}\ \mathit{ws}\ (\mathit{unC}_7\ x\ h)))
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathit{width}_7 :: \mathit{Circuit}_7 \to \mathit{Width}\\
&\mathit{width}_7\ \mathit{circuit} = \mathit{unC}_7\ \mathit{circuit}\ \mathit{widthAlg}\\
&\mathit{depth}_7 :: \mathit{Circuit}_7 \to \mathit{Depth}\\
&\mathit{depth}_7\ \mathit{circuit} = \mathit{unC}_7\ \mathit{circuit}\ \mathit{depthAlg}
\end{aligned}
$$

In fact, the shallow embedding provides a universal generic interpretation as the *Church encoding* [15] of the AST—or more precisely, because it is typed, the *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]:

$$
\begin{aligned}
&\mathit{deep} :: \mathit{Circuit}_7 \to \mathit{Circuit}_4\\
&\mathit{deep}\ \mathit{circuit} = \mathit{unC}_7\ \mathit{circuit}\ \mathit{In}
\end{aligned}
$$

(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:

$$
\begin{aligned}
&\mathit{shallow} :: \mathit{Circuit}_4 \to \mathit{Circuit}_7\\
&\mathit{shallow} = \mathit{foldC}\ \mathit{shallowAlg}\\
&\mathit{shallowAlg} :: \mathit{CircuitAlg}\ \mathit{Circuit}_7\\
&\mathit{shallowAlg}\ (\mathit{IdentityF}\ w) &&= \mathit{identity}_7\ w\\
&\mathit{shallowAlg}\ (\mathit{FanF}\ w) &&= \mathit{fan}_7\ w\\
&\mathit{shallowAlg}\ (\mathit{AboveF}\ c\ d) &&= \mathit{above}_7\ c\ d\\
&\mathit{shallowAlg}\ (\mathit{BesideF}\ c\ d) &&= \mathit{beside}_7\ c\ d\\
&\mathit{shallowAlg}\ (\mathit{StretchF}\ \mathit{ws}\ c) &&= \mathit{stretch}_7\ \mathit{ws}\ c
\end{aligned}
$$

Moreover, *deep* and *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:

$$
\begin{aligned}
&\mathbf{class}\ \mathit{Circuit}_8\ \mathit{circuit}\ \mathbf{where}\\
&\quad \mathit{identity}_8 :: \mathit{Size} \to \mathit{circuit}\\
&\quad \mathit{fan}_8 :: \mathit{Size} \to \mathit{circuit}\\
&\quad \mathit{above}_8 :: \mathit{circuit} \to \mathit{circuit} \to \mathit{circuit}\\
&\quad \mathit{beside}_8 :: \mathit{circuit} \to \mathit{circuit} \to \mathit{circuit}\\
&\quad \mathit{stretch}_8 :: [\mathit{Size}] \to \mathit{circuit} \to \mathit{circuit}
\end{aligned}
$$

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:

$$\mathbf{newtype}\ \mathit{Width}_8 = \mathit{Width}\ \{\mathit{unWidth} :: \mathit{Int}\}$$

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Circuit}_8\ \mathit{Width}_8\ \textbf{where} \\
\quad \mathit{identity}_8\ w \quad = \mathit{Width}\ w \\
\quad \mathit{fan}_8\ w \qquad = \mathit{Width}\ w \\
\quad \mathit{above}_8\ x\ y \quad = x \\
\quad \mathit{beside}_8\ x\ y \quad = \mathit{Width}\ (\mathit{unWidth}\ x + \mathit{unWidth}\ y) \\
\quad \mathit{stretch}_8\ \mathit{ws}\ x = \mathit{Width}\ (\mathit{sum}\ \mathit{ws})
\end{array}
$$

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:

$$
\begin{array}{l}
\textbf{newtype}\ \mathit{Depth}_8 = \mathit{Depth}\ \{\mathit{unDepth} :: \mathit{Int}\} \\
\textbf{instance}\ \mathit{Circuit}_8\ \mathit{Depth}_8\ \textbf{where} \\
\quad \mathit{identity}_8\ w \quad = \mathit{Depth}\ 0 \\
\quad \mathit{fan}_8\ w \qquad = \mathit{Depth}\ 1 \\
\quad \mathit{above}_8\ x\ y \quad = \mathit{Depth}\ (\mathit{unDepth}\ x + \mathit{unDepth}\ y) \\
\quad \mathit{beside}_8\ x\ y \quad = \mathit{Depth}\ (\mathit{unDepth}\ x\ \text{`}\mathit{max}\text{`}\ \mathit{unDepth}\ y) \\
\quad \mathit{stretch}_8\ \mathit{ws}\ x = x
\end{array}
$$

Some of the wrapping and unwrapping of Width_8 and Depth_8 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.

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 ($\mathit{identity}_8, \mathit{fan}_8$ etc). For an instance c of the type class $\mathit{Circuit}_8$, the dictionary type is equivalent to *CircuitAlg c*. Indeed, we might have defined instead

$$
\begin{array}{l}
\textbf{class}\ \mathit{Circuit}_9\ c\ \textbf{where} \\
\quad \mathit{alg} :: \mathit{CircuitAlg}\ c \\
\textbf{instance}\ \mathit{Circuit}_9\ \mathit{Width}_8\ \textbf{where} \\
\quad \mathit{alg} = \mathit{Width} \circ \mathit{widthAlg} \circ \mathit{fmap}\ \mathit{unWidth}
\end{array}
$$

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 in the everyday language by translation to the core language.

For example, the *identity* construct in our DSL of circuits is redundant: *identity* 1 is morally equivalent to *fan* 1, and for any other width n, we can construct a circuit equivalent to *identity n* by placing n copies of *identity* 1 side by side (or alternatively, as *stretch* $[n]$ (*identity* 1)). One might therefore identify a simpler datatype

$$
\begin{array}{llll}
\multicolumn{4}{l}{\textbf{data}\ \mathit{CoreCircuit} :: * \ \textbf{where}} \\
\quad \mathit{CFan} & :: \mathit{Size} \to & & \mathit{CoreCircuit} \\
\quad \mathit{CAbove} & :: \mathit{CoreCircuit} \to & \mathit{CoreCircuit} \to & \mathit{CoreCircuit} \\
\quad \mathit{CBeside} & :: \mathit{CoreCircuit} \to & \mathit{CoreCircuit} \to & \mathit{CoreCircuit} \\
\quad \mathit{CStretch} & :: [\mathit{Size}] \to & \mathit{CoreCircuit} \to & \mathit{CoreCircuit}
\end{array}
$$

and use it as the carrier of a shallow embedding for the everyday language. The everyday constructs that correspond to core constructs are represented directly; the derived constructs are defined by translation.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textbf{type}\ \mathit{Circuit}_{10} = \mathit{CoreCircuit}} \\
\multicolumn{2}{l}{\mathit{coreAlg} :: \mathit{CircuitAlg}\ \mathit{Circuit}_{10}} \\
\mathit{coreAlg}\ (\mathit{IdentityF}\ w) & = \mathit{foldr1}\ \mathit{CBeside}\ (\mathit{replicate}\ w\ (\mathit{CFan}\ 1)) \\
\mathit{coreAlg}\ (\mathit{FanF}\ w) & = \mathit{CFan}\ w \\
\mathit{coreAlg}\ (\mathit{AboveF}\ x\ y) & = \mathit{CAbove}\ x\ y \\
\mathit{coreAlg}\ (\mathit{BesideF}\ x\ y) & = \mathit{CBeside}\ x\ y \\
\mathit{coreAlg}\ (\mathit{StretchF}\ \mathit{ws}\ x) & = \mathit{CStretch}\ \mathit{ws}\ x
\end{array}
$$

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:

$$
\mathit{rstretch}\ (\mathit{ws} +\!\!+ [w+1])\ c = \mathit{stretch}\ (1 : \mathit{ws})\ c\ \text{`}\mathit{beside}\text{`}\ \mathit{identity}\ w
$$

So one might extend the grammar of the everyday language, as embodied in the functor *CircuitF* or the type class $\mathit{Circuit}_8$, 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:

$$
\begin{array}{lll}
\textbf{data}\ \mathit{Identity}_{11}\ c & = \mathit{Identity}_{11}\ \mathit{Size} & \textbf{deriving}\ \mathit{Functor} \\
\textbf{data}\ \mathit{Fan}_{11}\ c & = \mathit{Fan}_{11}\ \mathit{Size} & \textbf{deriving}\ \mathit{Functor} \\
\textbf{data}\ \mathit{Above}_{11}\ c & = \mathit{Above}_{11}\ c\ c & \textbf{deriving}\ \mathit{Functor} \\
\textbf{data}\ \mathit{Beside}_{11}\ c & = \mathit{Beside}_{11}\ c\ c & \textbf{deriving}\ \mathit{Functor} \\
\textbf{data}\ \mathit{Stretch}_{11}\ c & = \mathit{Stretch}_{11}\ [\mathit{Size}]\ c & \textbf{deriving}\ \mathit{Functor}
\end{array}
$$

with a right-associating 'sum' operator for combining them:

$$
\begin{array}{l}
\textbf{data}\ (f :+: g)\ e = \mathit{Inl}\ (f\ e) \mid \mathit{Inr}\ (g\ e)\ \textbf{deriving}\ \mathit{Functor} \\
\textbf{infixr} :+:
\end{array}
$$

One can assemble a functor from these components and make a deep embedding from it. For example, the sum of functors $\mathit{CircuitF}_{11}$ is equivalent to *CircuitF* from the start of Section 4, and its fixpoint $\mathit{Circuit}_{11}$ to $\mathit{Circuit}_4$:

$$
\begin{array}{l}
\textbf{type}\ \mathit{CircuitF}_{11} = \mathit{Identity}_{11} :+: \mathit{Fan}_{11} :+: \mathit{Above}_{11} :+: \\
\qquad\qquad\qquad\quad \mathit{Beside}_{11} :+: \mathit{Stretch}_{11} \\
\textbf{data}\ \mathit{Fix}\ f = \mathit{In}\ (f\ (\mathit{Fix}\ f)) \\
\textbf{type}\ \mathit{Circuit}_{11} = \mathit{Fix}\ \mathit{CircuitF}_{11}
\end{array}
$$

This works, but it is rather clumsy. In particular, an expression of type $\mathit{Circuit}_{11}$ involves a mess of *Inl*, *Inr* and *In* constructors, as seen in this rendition of the circuit in Figure 4:

```
stretchfan :: Circuit_11
stretchfan = In (Inr (Inr (Inr (Inr (Stretch_11 [3,2,3] (
               In (Inr (Inl (Fan_11 3)))))))))
```

Fortunately, there is an obvious way of injecting payloads into sum types in this fashion, which we can express through a simple notion of subtyping between functors, witnessed by an injection:

```
class (Functor f, Functor g) ⇒ f :≺: g where
  inj :: f a → g a
```

Subtyping is reflexive, and summands are subtypes of their sum:

```
instance Functor f ⇒ f :≺: f where
  inj = id
instance (Functor f, Functor g) ⇒ f :≺: (f :+: g) where
  inj = Inl
instance (Functor f, Functor g, Functor h, f :≺: g) ⇒
           f :≺: (h :+: g) where
  inj = Inr ∘ inj
```

Note that these type class instances overlap, going beyond Haskell 98; nevertheless, as Swierstra explains, provided that sums are associated to the right this should not cause any problems.

Now we can define smart constructors that inject in this 'obvious' way:

```
identity_11 :: (Identity_11 :≺: f) ⇒ Width → Fix f
identity_11 w = In (inj (Identity_11 w))
fan_11 :: (Fan_11 :≺: f) ⇒ Width → Fix f
fan_11 w = In (inj (Fan_11 w))
above_11 :: (Above_11 :≺: f) ⇒ Fix f → Fix f → Fix f
above_11 x y = In (inj (Above_11 x y))
beside_11 :: (Beside_11 :≺: f) ⇒ Fix f → Fix f → Fix f
beside_11 x y = In (inj (Beside_11 x y))
stretch_11 :: (Stretch_11 :≺: f) ⇒ [Width] → Fix f → Fix f
stretch_11 ws x = In (inj (Stretch_11 ws x))
```

and the mess of injections can be inferred instead:

```
stretchfan :: (Fan_11 :≺: f, Stretch_11 :≺: f) ⇒ Fix f
stretchfan = stretch_11 [3,2,3] (fan_11 3)
```

Crucially, this technique also leaves the precise choice of grammar open; all that is required is for the grammar to provide fan and stretch constructors, and we can capture that dependence in the flexible declared type for *stretchfan*.

Interpretations can be similarly modularized. Of course, we expect them to be folds:

```
fold :: Functor f ⇒ (f a → a) → Fix f → a
fold h (In x) = h (fmap (fold h) x)
```

In order to accommodate open datatypes, we define interpretations in pieces. We declare a type class of those constructors supporting a given interpretation:

```
class Functor f ⇒ WidthAlg f where
  widthAlg_11 :: f Width → Width
```

Interpretations lift through sums in the obvious way:

```
instance (WidthAlg f, WidthAlg g) ⇒ WidthAlg (f :+: g) where
  widthAlg_11 (Inl x) = widthAlg_11 x
  widthAlg_11 (Inr y) = widthAlg_11 y
```

Then we provide instances for each of the relevant constructors. For example, if we only ever wanted to compute the width of circuits expressed in terms of the fan and stretch constructors, we need only define those two instances:

```
instance WidthAlg Fan_11 where
  widthAlg_11 (Fan_11 w) = w
instance WidthAlg Stretch_11 where
  widthAlg_11 (Stretch_11 ws x) = sum ws
```

For example, this width function works for the flexibly typed circuit *stretchfan* above:

```
width_11 :: WidthAlg f ⇒ Fix f → Width
width_11 = fold widthAlg_11
```

—although the circuit does need to be given a specific type first:

```
> width_11 (stretchfan :: Circuit_11)
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 RStretch_11 c = RStretch_11 [Size] c deriving Functor
rstretch_11 :: (RStretch_11 :≺: f) ⇒ [Width] → Fix f → Fix f
rstretch_11 ws x = In (inj (RStretch_11 ws x))
```

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

```
rstretchfan :: (Identity_11 :≺: f, Fan_11 :≺: f, Beside_11 :≺: f,
                RStretch_11 :≺: f) ⇒ Fix f
rstretchfan = beside_11 (identity_11 2)
                        (rstretch_11 [2,3,1] (fan_11 3))
```

When adding new constructors such as $\mathit{RStretch}_{11}$, 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 $\mathit{rstretch}_{11}$ can itself be simulated out of existing components. We might instead write a function that *handles* the $\mathit{RStretch}_{11}$ constructor:

```
handle :: (Stretch_11 :≺: f, Beside_11 :≺: f, Identity_11 :≺: f) ⇒
            Fix (RStretch_11 :+: f) → Fix f
handle (In (Inl (RStretch_11 ws c))) =
  stretch_11 (1 : ws') (handle c) `beside_11` identity_11 w
    where (ws', w) = (init ws, last ws − 1)
handle (In (Inr other)) = In (fmap handle other)
```

Here, we recursively translate all instances of $\mathit{RStretch}_{11}$ 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 $\mathit{RStretch}_{11}$ constructors before passing the result on to an interpretation function such as width_{11} that need not deal with $\mathit{RStretch}_{11}$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 $\mathit{Circuit}_4$) and their Böhm–Berarducci encodings (such as $\mathit{Circuit}_7$) are equivalent. And of course, none of this is specific in any way to the *Circuit* DSL; a datatype-generic version of the story can be told, by abstracting away from the shape functor *CircuitF*—the reader may enjoy working out the details.

Nevertheless, the observation that shallow embeddings correspond 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 corollaries. The three generalizations of folds (banana split, mutumorphisms, and accumulating parameters) exploited in Section 4 are all special cases of *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 McKinna, and the anonymous reviewers all made helpful comments, and Oleg Kiselyov gave many constructive criticisms and much inspiration, 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

[1] Robert Atkey, Sam Lindley, and Jeremy Yallop. Unembedding domain-specific languages. In *Haskell Symposium*, pages 37–48. ACM, 2009.

[2] Lennart Augustsson. Making EDSLs fly. In *TechMesh*, London, December 2012. Video at `http://vimeo.com/73223479`.

[3] Richard S. Bird. The promotion and accumulation strategies in transformational programming. *ACM Transactions on Programming Languages and Systems*, 6(4):487–504, October 1984. Addendum in TOPLAS 7(3):490–492, July 1985.

[4] Corrado Böhm and Alessandro Berarducci. Automatic synthesis of typed λ-programs on term algebras. *Theoretical Computer Science*, 39:135–154, 1985.

[5] Richard Boulton. Personal communication, 10th February 2014.

[6] Richard Boulton, Andrew Gordon, Mike Gordon, John Harrison, John Herbert, and John Van Tassel. Experience with embedding hardware description languages in HOL. In Victoria Stavridou, Thomas F. Melham, and Raymond T. Boute, editors, *IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, pages 129–156. North-Holland/Elsevier, 1992.

[7] Richard P. Brent and Hsiang-Tsung Kung. The chip complexity of binary arithmetic. In *Symposium on Theory of Computing*, pages 190–200. ACM, 1980.

[8] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(5):509–543, 2009.

[9] Haskell B. Curry, J. Roger Hindley, and Jonathan P. Seldin. *Combinatory Logic*, volume 2. North-Holland, 1972.

[10] Maarten M. Fokkinga. Tupling and mutumorphisms. *The Squiggolist*, 1(4):81–82, June 1990.

[11] Martin Fowler. *Domain-Specific Languages*. Addison-Wesley, 2011.

[12] Jeremy Gibbons. Functional programming for domain-specific languages. In Viktória Zsók, editor, *Central European Functional Programming Summer School*, volume 8606 of *Lecture Notes in Computer Science*, pages 1–27. Springer, 2014. To appear.

[13] Jeremy Gibbons and Geraint Jones. The under-appreciated unfold. In *International Conference on Functional Programming*, pages 273–279, Baltimore, Maryland, September 1998.

[14] Ralf Hinze. An algebra of scans. In *Mathematics of Program Construction*, volume 3125 of *Lecture Notes in Computer Science*, pages 186–210. Springer, 2004.

[15] Ralf Hinze. Church numerals, twice! *Journal of Functional Programming*, 15(1), 2005.

[16] Ralf Hinze. Adjoint folds and unfolds: An extended study. *Science of Computer Programming*, 78(11):2108–2159, 2013.

[17] Ralf Hinze, Nicolas Wu, and Jeremy Gibbons. Unifying structured recursion schemes. In *International Conference on Functional Programming*, pages 209–220, Boston, Massachusetts, September 2013.

[18] Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. Formal derivation of efficient parallel programs by construction of list homomorphisms. *ACM Transactions on Programming Languages and Systems*, 19(3):444–461, 1997.

[19] Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4), 1996.

[20] Anne Kaldewaij. *Programming: The Derivation of Algorithms*. Prentice Hall, 1990.

[21] Oleg Kiselyov. Typed tagless final interpreters. In Jeremy Gibbons, editor, *Generic and Indexed Programming*, volume 7470 of *Lecture Notes in Computer Science*, pages 130–174. Springer, 2012.

[22] Oleg Kiselyov, Amr Sabry, and Cameron Swords. Extensible effects: An alternative to monad transformers. In *Haskell Symposium*, pages 59–70. ACM, 2013.

[23] Shriram Krishnamurthi, Matthias Felleisen, and Daniel P. Friedman. Synthesizing object-oriented and functional design to promote re-use. In *European Conference on Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 91–113. Springer, 1998.

[24] Ehud Lamm. CUFP write-up. Blog post, `http://lambda-the-ultimate.org/node/2572`, December 2007.

[25] Ralf Lämmel, Joost Visser, and Jan Kort. Dealing with large bananas. In Johan Jeuring, editor, *Workshop on Generic Programming*, volume Technical Report UU-CS-2000-19. Universiteit Utrecht, 2000.

[26] John Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. In Stephen A. Schuman, editor, *New Directions in Algorithmic Languages*, pages 157–168, 1975.

[27] Josef Svenningsson and Emil Axelsson. Combining deep and shallow embedding for embedded domain-specific languages. In *Trends in Functional Programming 2012*, volume 7829 of *Lecture Notes in Computer Science*, pages 21–36, 2013.

[28] Wouter Swierstra. Datatypes à la carte. *Journal of Functional Programming*, 18(4):423–436, 2008.

[29] Philip Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture*, pages 347–359. ACM, 1989.

[30] Philip Wadler. The expression problem. Java Genericity Mailing list, November 1998. `http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt`.

[31] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Principles of Programming Languages*, pages 60–76. ACM, 1989.