Folding Domain-Specific Languages: Deep and Shallow Embeddings

Jeremy Gibbons
www.cs.ox.ac.uk/jeremy.gibbons
University of Oxford

October 2013

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 *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 variables, definitions, and conditionals.

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 for the host language can be carried over to the DSL. Whereas the standalone approach is the most common one within object-oriented circles [5], the embedded approach is typically favoured by functional programmers [11]. 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 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); this tree is subsequently transformed for optimization and traversed for evaluation. With a *shallow embedding*, terms in the DSL are implemented directly as the values to which they evaluate, bypassing the intermediate AST and its traversal. The deep and shallow embeddings are closely related, and intimately connected to folds; the purpose of this paper is to explore that connection.

2 Expressions

Consider a very simple language of arithmetic expressions, involving integer constants and addition. As a deeply embedded DSL, this can be captured by the following algebraic datatype:

$$
\begin{array}{l}
\mathbf{data}\ \mathit{Expr} :: * \ \mathbf{where} \\
\quad \mathit{Val}\ \ :: \mathit{Integer} \rightarrow \mathit{Expr} \\
\quad \mathit{Add} :: \mathit{Expr} \rightarrow \mathit{Expr} \rightarrow \mathit{Expr}
\end{array}
$$

(We have used Haskell's 'generalized algebraic datatype' notation, in order to make the types of the constructors *Val* and *Add* explicit; but we are not using the generality of GADTs here, and the old-fashioned way would have worked too.) The expression $3 + 4$ is represented by the term $\mathit{Add}\ (\mathit{Val}\ 3)\ (\mathit{Val}\ 4)$ in the DSL. Observations of terms in the DSL are defined as functions over the algebraic datatype; for example, here is how to evaluate an expression:

$$
\begin{array}{l}
\mathit{eval} :: \mathit{Expr} \rightarrow \mathit{Integer} \\
\mathit{eval}\ (\mathit{Val}\ n) \qquad = n \\
\mathit{eval}\ (\mathit{Add}\ x\ y) = \mathit{eval}\ x + \mathit{eval}\ y
\end{array}
$$

A shallow embedding eschews the algebraic datatype, which records 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:

$$
\begin{array}{l}
\mathbf{type}\ \mathit{Expr} = \mathit{Integer} \\[4pt]
\mathit{val} :: \mathit{Integer} \rightarrow \mathit{Expr} \\
\mathit{val}\ n \quad\ = n \\[4pt]
\mathit{add} :: \mathit{Expr} \rightarrow \mathit{Expr} \rightarrow \mathit{Expr} \\
\mathit{add}\ x\ y = x + y
\end{array}
$$

One might see the deep and shallow embeddings as duals, in a variety of senses. For one sense, the language constructs *Val* 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 *val* and *add* do all the work, and the observer (of type $\mathit{Expr} \rightarrow \mathit{Integer}$) is simply the identity function and so is omitted.

For a second sense, it is trivial to add a second observer to the deep embedding—just define another function alongside *eval*—but awkward to add new constructs: doing so entails revisiting the definitions of all exisiting observers to add an additional clause. In contrast, adding a construct to the shallow embedding is trivial—alongside *val* and *add*—but introducing an additional observer entails completely revising the semantics by changing the definitions of all existing constructs. This tension is precisely the conflict of forces addressed by the VISITOR design pattern in object-oriented programming [6].

The types of *val* and *add* in the shallow embedding coincide with those of *Val* and *Add* in the deep embedding; moreover, the definitions of *val* 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. In other words, *shallow embeddings correspond to folds over the abstract syntax captured by a deep embedding.*

Note that we do not claim duality in the categorical sense of reversing arrows. Similarly, deep and shallow embeddings have been called the 'initial' and 'final' approaches [3], but only in an informal sense; in fact, the two approaches both correspond to initial algebras, and neither to final coalgebras.

3 Folds

Folds are the natural pattern of computation induced by 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 *Expr* above has two constructors; *Val* takes one argument, of the fixed type *Integer*; *Add* takes two arguments, both recursive occurrences. Thus, we rule out contravariant recursion, polymorphic datatypes, higher kinds, and other such esoterica.

The general case is captured by a shape, an instance of the *Functor* type class:

$$
\begin{array}{l}
\mathbf{class}\ \mathit{Functor}\ f\ \mathbf{where} \\
\quad \mathit{fmap} :: (a \rightarrow b) \rightarrow (f\ a \rightarrow f\ b)
\end{array}
$$

For Expr, the shape is as follows:

$$
\begin{array}{l}
\textbf{data}\ \mathit{ExprF} :: * \to * \ \textbf{where} \\
\quad \mathit{ValF}\ \ :: \mathit{Integer} \to \mathit{ExprF}\ a \\
\quad \mathit{AddF} :: a \to a\ \ \to \mathit{ExprF}\ a
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Functor}\ \mathit{ExprF}\ \textbf{where} \\
\quad \mathit{fmap}\ f\ (\mathit{ValF}\ n) \qquad = \mathit{ValF}\ n \\
\quad \mathit{fmap}\ f\ (\mathit{AddF}\ x\ y) = \mathit{AddF}\ (f\ x)\ (f\ y)
\end{array}
$$

For a given functor such as ExprF expressing a language shape, the deeply embedded DSL of that shape is the so-called initial algebra of the functor:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Deep} :: (* \to *) \to * \ \textbf{where} \\
\quad \mathit{In} :: \mathit{Functor}\ f \Rightarrow f\ (\mathit{Deep}\ f) \to \mathit{Deep}\ f
\end{array}
$$

$$
\textbf{type}\ \mathit{Expr} = \mathit{Deep}\ \mathit{ExprF}
$$

Compositional interpretations are precisely the folds for these initial algebras, morphisms to other algebras for the functor f:

$$
\textbf{type}\ \mathit{Algebra}\ f\ a = f\ a \to a
$$

$$
\begin{array}{l}
\mathit{fold} :: \mathit{Functor}\ f \Rightarrow \mathit{Algebra}\ f\ a \to \mathit{Deep}\ f \to a \\
\mathit{fold}\ \mathit{phi}\ (\mathit{In}\ x) = \mathit{phi}\ (\mathit{fmap}\ (\mathit{fold}\ \mathit{phi})\ x)
\end{array}
$$

For example, eval is a fold for the deeply embedded DSL of shape ExprF:

$$
\begin{array}{l}
\mathit{evalAlg} :: \mathit{Algebra}\ \mathit{ExprF}\ \mathit{Integer} \\
\mathit{evalAlg}\ (\mathit{ValF}\ n) \qquad = n \\
\mathit{evalAlg}\ (\mathit{AddF}\ x\ y) = x + y
\end{array}
$$

$$
\begin{array}{l}
\mathit{eval} :: \mathit{Expr} \to \mathit{Integer} \\
\mathit{eval} = \mathit{fold}\ \mathit{evalAlg}
\end{array}
$$

The shallow embedding is simply the algebra, such as $\mathit{evalAlg}$. So an observation function for the deep embedding, such as eval, is precisely a fold using the shallow embedding as the algebra. This insight is very revealing: we know a lot about folds, and this tells us a lot about embedded DSLs. We discuss these consequences next.

3.1 Multiple interpretations

As mentioned above, the deep embedding smoothly supports additional observations. For example, suppose that we also wanted to print expressions as strings; no problem—we can just define another observation function print.

$$
\begin{array}{l}
\mathit{print} :: \mathit{Expr} \to \mathit{String} \\
\mathit{print}\ (\mathit{Val}\ n) \qquad = \mathit{show}\ n \\
\mathit{print}\ (\mathit{Add}\ x\ y) = \mathit{paren}\ (\mathit{print}\ x +\!\!+ \texttt{"+"} +\!\!+ \mathit{print}\ y)
\end{array}
$$

where we define for later reuse

$$paren\ s = \texttt{"("} + s + \texttt{")"}$$

But what about a shallow embedding? With this approach, expressions can only have a single semantics, so how do we accommodate both evaluation and printing? It's not difficult; we simply make the semantics a pair, providing both interpretations simultaneously, so that the observation functions *eval* and *print* become projections rather than just the identity function.

$$\begin{array}{l}
\mathbf{type}\ Expr = (Integer, String) \\
val :: Integer \rightarrow Expr \\
val\ n \quad = (n, show\ n) \\
add :: Expr \rightarrow Expr \rightarrow Expr \\
add\ x\ y = (eval\ x + eval\ y, paren\ (print\ x + \texttt{"+"} + print\ y)) \\
eval :: Expr \rightarrow Integer \\
eval = fst \\
print :: Expr \rightarrow String \\
print = snd
\end{array}$$

Of course, this works best under lazy evaluation: if only one of the two interpretations on an expression is needed, only that one is evaluated.

Seen from the fold perspective, this step is no surprise: the 'banana split law' [4] 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.

3.2 Strengthening the invariant

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 from the interpretations of its parts; it is both a valuable property for reasoning and a significant limitation to expressivity.

For example, recall the *print* interpretation above, which produces a fully parenthesized string; suppose one wanted a slightly more sophisticated rendering instead, using parentheses only where necessary to capture the structure of the expression. (Of course, addition is associative, so the parenthesization of an expression makes no difference to its value; but value isn't everything.) The function *mprint* 'minimally prints' an expression, only parenthesizing subexpressions, and then only if they are *Add*s rather than *Val*s:

$$mprint\ (Add\ (Val\ 3)\ (Add\ (Val\ 4)\ (Val\ 5))) = \texttt{"3+(4+5)"}$$

We have:

$$
\begin{array}{l}
mprint :: Expr \to String \\
mprint\ (Val\ n) \quad\ \ = show\ n \\
mprint\ (Add\ x\ y) = pprint\ x \mathbin{+\!\!+} \texttt{"+"} \mathbin{+\!\!+} pprint\ y \\
\quad \textbf{where} \\
\qquad pprint\ e = (\textbf{if}\ isAdd\ e\ \textbf{then}\ paren\ \textbf{else}\ id)\ (mprint\ e) \\
isAdd\ (Val\ \_) \quad\ \ = False \\
isAdd\ (Add\ \_\ \_) = True
\end{array}
$$

This is a non-compositional interpretation of the abstract syntax, because *mprint* depends on *isAdd* of subexpressions as well as their recursive image under *mprint*. (True, you can reconstruct *isAdd e* from *mprint e*, if you try hard enough. But in general, one interpretation might depend on a second that is not derivable from the first, as for example the average of a list depends on both its sum and its length, neither of which is derivable from the other.)

What can we do about such non-compositional interpretations in the shallow embedding? Again, fold theory comes to the rescue: *mprint* and *isVal* together form a *mutumorphism* [4]—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* [4], since the dependency is only one-way; and a particularly simple example of a zygomorphism at that, because *isVal* is a trivial non-recursive fold. Simpler still, in the banana split above, neither of the two folds depends on the other.)

$$
\begin{array}{l}
\textbf{type}\ Expr = (String, Bool) \\
val :: Integer \to Expr \\
val\ n \quad\ = (show\ n, False) \\
add :: Expr \to Expr \to Expr \\
add\ x\ y = (pprint\ x \mathbin{+\!\!+} \texttt{"+"} \mathbin{+\!\!+} pprint\ y, True) \\
\quad \textbf{where} \\
\qquad pprint\ e = (\textbf{if}\ isAdd\ e\ \textbf{then}\ paren\ \textbf{else}\ id)\ (mprint\ e) \\
mprint :: Expr \to String \\
mprint = fst \\
isAdd :: Expr \to Bool \\
isAdd = snd
\end{array}
$$

Tupling functions in this way is analogous to strengthening the invariant of an imperative loop to record additional information [12], and is a standard trick in program calculation [10]. For example, when solving the 'maximum segment sum' problem [1] as a loop, one strengthens the invariant that s is the maximum segment sum seen so far:

$$s = (\mathsf{max}\ j, k\ :\ 0 \leqslant j \leqslant k < n\ :\ (\mathsf{sum}\ i\ :\ j \leqslant i < k\ :\ a\,[i])) \qquad (*)$$

by adding the conjunct that t is the maximum suffix sum seen so far:

$$t = (\mathsf{max}\ j\ :\ 0 \leqslant j < n\ :\ (\mathsf{sum}\ i\ :\ j \leqslant i < n\ :\ a\,[i])) \qquad (**)$$

Conjoining the main invariant ($*$) with the auxilliary invariant ($**$) is analogous to tupling the 'maximum segment sum' function with the 'maximum suffix sum' function.

3.3 Context-sensitivity

Another way of achieving minimal parenthesization is to print expressions differently depending on their context: *Add* subexpressions that are themselves arguments in enclosing expressions should be parenthesized, but outermost *Add* expressions and all *Val* expressions should not. More generally, expressions might be constructed from many different operators with different precedences, and an inner expression should be parenthesized iff it has lower precedence than the enclosing operator.

This too can be achieved using standard fold techniques. We use an *accumulating parameter* to carry the context into a subexpression:

$$
\begin{array}{l}
\mathit{mprint} :: \mathit{Expr} \rightarrow \mathit{String} \\
\mathit{mprint}\ e = \mathit{cprint}\ \mathit{False}\ e \\[4pt]
\mathit{cprint} :: \mathit{Bool} \rightarrow \mathit{Expr} \rightarrow \mathit{String} \\
\mathit{cprint}\ \_\ (\mathit{Val}\ n) \quad\ \ = \mathit{show}\ n \\
\mathit{cprint}\ b\ (\mathit{Add}\ x\ y) = (\mathbf{if}\ b\ \mathbf{then}\ \mathit{paren}\ \mathbf{else}\ \mathit{id})\ (\mathit{cprint}\ \mathit{True}\ x + \texttt{"+"} + \mathit{cprint}\ \mathit{True}\ y)
\end{array}
$$

Now, *cprint* is not a fold, because *cprint b* of an *Add* expression depends on a different function *cprint True* of its children; but *flip cprint* is a fold, yielding a result of type $\mathit{Bool} \rightarrow \mathit{String}$, so we can use this as the semantics in a shallow embedding.

$$
\begin{array}{l}
\mathbf{type}\ \mathit{Expr} = \mathit{Bool} \rightarrow \mathit{String} \\[4pt]
\mathit{val} :: \mathit{Integer} \rightarrow \mathit{Expr} \\
\mathit{val}\ n\ \_ \quad = \mathit{show}\ n \\[4pt]
\mathit{add} :: \mathit{Expr} \rightarrow \mathit{Expr} \rightarrow \mathit{Expr} \\
\mathit{add}\ x\ y\ b = (\mathbf{if}\ b\ \mathbf{then}\ \mathit{paren}\ \mathbf{else}\ \mathit{id})\ (x\ \mathit{True} + \texttt{"+"} + y\ \mathit{True}) \\[4pt]
\mathit{mprint} :: \mathit{Expr} \rightarrow \mathit{String} \\
\mathit{mprint}\ e = e\ \mathit{False}
\end{array}
$$

A similar technique could be used for *eval*, if we wanted to introduce variable references and 'let' expressions into the language; the interpretation would be as functions from environments to values, extending the environment as 'let' bindings are encountered.

3.4 Generic interpretation

We have seen 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 ten-tuples are never pleasant to work with.

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

$$
\begin{aligned}
&\textbf{type}\ \mathit{Expr} = \forall a\,.\ \mathit{Algebra}\ \mathit{ExprF}\ a \to a\\
&\mathit{val} :: \mathit{Integer} \to \mathit{Expr}\\
&\mathit{val}\ n\ \mathit{phi} \quad = \mathit{phi}\ (\mathit{ValF}\ n)\\
&\mathit{add} :: \mathit{Expr} \to \mathit{Expr} \to \mathit{Expr}\\
&\mathit{add}\ x\ y\ \mathit{phi} = \mathit{phi}\ (\mathit{AddF}\ (x\ \mathit{phi})\ (y\ \mathit{phi}))
\end{aligned}
$$

That this encoding is generic is evidenced by the fact that it can be instantiated to yield evaluation and printing (and, of course, any other fold):

$$
\begin{aligned}
&\mathit{eval} :: \mathit{Expr} \to \mathit{Integer}\\
&\mathit{eval}\ e \quad = e\ \mathit{evalAlg}\\
&\mathit{print} :: \mathit{Expr} \to \mathit{String}\\
&\mathit{print}\ e = e\ \mathit{printAlg}\\
&\quad \textbf{where}\\
&\qquad \mathit{printAlg} :: \mathit{Algebra}\ \mathit{ExprF}\ \mathit{String}\\
&\qquad \mathit{printAlg}\ (\mathit{ValF}\ n) \quad\ = \mathit{show}\ n\\
&\qquad \mathit{printAlg}\ (\mathit{AddF}\ x\ y) = \mathit{paren}\ (x +\!\!+ \texttt{"+"} +\!\!+ y)
\end{aligned}
$$

In fact, the shallow embedding provides a universal generic interpretation as the *Church encoding* [2, 8] of the AST.

4 Discussion

The essential observation made here—that shallow embeddings correspond to folds over the abstract syntax captured by a deep embedding—is surely not new. For example, it was probably known to Reynolds [14], 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 [3], 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 taglessness. (The observation is more explicit in Kiselyov’s lecture notes on the finally tagless approach [13], which go into more detail on compositionality.)

Nevertheless, the observation seems not to be widely appreciated. 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 3 are all special cases of *adjoint fold* [9]; perhaps other adjoint folds yield more 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, and I thank the organizers for the invitation to lecture there. Nick Wu, Pedro Magalhaes, and Ralf Hinze in the Algebra of Programming group at Oxford made helpful comments, for which I am grateful. This is an earlier and shorter draft of a paper that was eventually published at ICFP 2014 [7].

References

[1] Jon Bentley. *Programming Pearls*. Addison-Wesley, 1986.

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

[3] 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.

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

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

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[7] Jeremy Gibbons and Nicolas Wu. Folding domain-specific languages: Deep and shallow embeddings. In *International Conference on Functional Programming*, September 2014.

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

[9] Ralf Hinze, Nicolas Wu, and Jeremy Gibbons. Unifying structured recursion schemes. In *ICFP*, 2013.

[10] 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.

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

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

[13] 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.

[14] 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.