Compositionality in embedded DSLs

(talk proposal)

Jeremy Gibbons

Department of Computer Science, University of Oxford

http://www.cs.ox.ac.uk/people/jeremy.gibbons/

1. Context

There are two main approaches to implementing domain-specific languages. With the *standalone* approach [1, 5], independent tools such as compilers and run-time environments for the DSL are implemented in one or more general-purpose programming languages. With the *embedded* approach [3, 8], the DSL implementation takes the form of a library of definitions in the host language, and a program in the DSL is merely a program in the host language that makes use of the library.

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. Deep embeddings might seem like the obvious approach, but Kamin [10] and Erwig [4] (among others) argue that shallow embeddings are superior.

Our focus in this talk proposal is the relationship between deep and shallow embeddings of DSLs, and the connection to compositional semantics.

2. Embeddings

Consider for example a DSL for 2D graphics, which might involve sublanguages for affine transformations (translation, scaling, rotation). As a deep embedding, this sublanguage could be represented using an algebraic datatype:

$$\begin{array}{l}
\textbf{data}\ \mathit{Transform}\ \textbf{where} \\
\quad \begin{array}{lll}
\mathit{Identity} & :: & \mathit{Transform} \\
\mathit{Translate} & :: \mathit{Complex} \rightarrow & \mathit{Transform} \\
\mathit{Scale} & :: \mathit{Real} \rightarrow & \mathit{Transform} \\
\mathit{Rotate} & :: \mathit{Real} \rightarrow & \mathit{Transform} \\
\mathit{Compose} & :: (\mathit{Transform}, \mathit{Transform}) \rightarrow & \mathit{Transform}
\end{array}
\end{array}$$

(The notation is Haskell; we trust that it is sufficiently self-explanatory.) This sublanguage might be used in various ways within the overall graphics language. For example, we probably want to transform points:

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathit{transform} :: (\mathit{Transform}, \mathit{Complex}) \rightarrow \mathit{Complex}} \\
\mathit{transform}\ (\mathit{Identity}, p) & = p \\
\mathit{transform}\ (\mathit{Translate}\ p, q) & = p + q \\
\mathit{transform}\ (\mathit{Scale}\ s, p) & = \mathit{scale}\ s\ p \\
\mathit{transform}\ (\mathit{Rotate}\ a, p) & = \mathit{rotate}\ a\ p \\
\mathit{transform}\ (\mathit{Compose}\ (t,u), p) & = \mathit{transform}\ (t, \mathit{transform}\ (u,p))
\end{array}$$

$$\begin{array}{l}
\mathit{scale}, \mathit{rotate} :: \mathit{Real} \rightarrow \mathit{Complex} \rightarrow \mathit{Complex} \\
\mathit{scale}\ s\ p\ \ = (s :+ 0) \times p \\
\mathit{rotate}\ a\ p = \mathit{mkPolar}\ (\mathit{magnitude}\ p)\ (a + \mathit{phase}\ p)
\end{array}$$

With a shallow embedding, we dispense with the abstract syntax trees (that is, the algebraic datatype), and represent terms directly by their semantics. This is straightforward to do for a single interpretation, for example representing a transformation directly as a function on points:

$$\begin{array}{ll}
\multicolumn{2}{l}{\textbf{type}\ \mathit{Transform}_S = \mathit{Complex} \rightarrow \mathit{Complex}} \\
\multicolumn{2}{l}{\mathit{identity}_S :: \mathit{Transform}_S} \\
\mathit{identity}_S & = \lambda p \rightarrow p \\
\multicolumn{2}{l}{\mathit{translate}_S :: \mathit{Complex} \rightarrow \mathit{Transform}_S} \\
\mathit{translate}_S\ p & = \lambda q \rightarrow p + q \\
\multicolumn{2}{l}{\mathit{scale}_S :: \mathit{Real} \rightarrow \mathit{Transform}_S} \\
\mathit{scale}_S\ s & = \lambda p \rightarrow \mathit{scale}\ s\ p \\
\multicolumn{2}{l}{\mathit{rotate}_S :: \mathit{Real} \rightarrow \mathit{Transform}_S} \\
\mathit{rotate}_S\ a & = \lambda p \rightarrow \mathit{rotate}\ a\ p \\
\multicolumn{2}{l}{\mathit{compose}_S :: (\mathit{Transform}_S, \mathit{Transform}_S) \rightarrow \mathit{Transform}_S} \\
\mathit{compose}_S\ (f,g) & = f \circ g
\end{array}$$

Note the similarity between the interpretation of a deep embedding (such as the clauses of *transform*) and the direct representation in a shallow embedding (via $\mathit{identity}_S$, $\mathit{translate}_S$, etc).

3. Tension

With a deep embedding, it is trivial to provide additional interpretations of a language:

$$\begin{array}{ll}
\mathit{isLinear} & :: \mathit{Transform} \rightarrow \mathit{Bool} \\
\mathit{print} & :: \mathit{Transform} \rightarrow \mathit{String}
\end{array}$$

But what about multiple interpretations in a shallow embedding? We don't want to have to redefine the representation *Transform* and reimplement all the constructors for each new interpretation. Sometimes we are lucky enough to have a common generalization of multiple interpretations (for example, we could represent a *Transform* as a matrix, and implement *transform* and *isLinear* in terms of this) but we are not always so lucky (it doesn't work for *print*). What is the general solution?

4. Resolution

The general pattern is that

> each feasible shallow embedding of a language corresponds to a *compositional* interpretation of the deep embedding of the language in question.

For example, the function *transform*—or rather, its curried version $\mathit{transform}' :: \mathit{Transform} \rightarrow (\mathit{Complex} \rightarrow \mathit{Complex})$—is compositional, in the sense that the interpretation $\mathit{transform}'\ (\mathit{Compose}\ (t,u))$ of a term $\mathit{Compose}\ (t,u)$ depends only on the interpretations $\mathit{transform}'\ t$ and $\mathit{transform}'\ u$ of its subterms t and u, and not on any other attributes of t and u. This is both a necessary and a sufficient condition for $transform'$ to be feasible as a shallow embedding of the language of transformations.

In functional programming, such compositional functions are called *folds* [9]. Folds follow a fixed recursion pattern, corresponding to the shape of the data structure; their points of variation constitute what is called an *algebra*, specifying how to interpret each constructor of the datatype. For example, the *Transform* datatype has five constructors, so an algebra for this shape of data is a quintuple, with one component per constructor:

$$\mathbf{type}\ TAlg\ a = (a, Complex \to a, Real \to a, Real \to a, (a,a) \to a)$$

The fold operator for *Transform*s takes such an algebra, and collapses a *Transform* down to a value; the recursion follows the shape of the *Transform*, and the individual constructors are handled by the corresponding components of the algebra:

$$\begin{array}{ll}
fold :: TAlg\ a \to (Transform \to a) & \\
fold\ (i,t,s,r,c)\ Identity & = i \\
fold\ (i,t,s,r,c)\ (Translate\ p) & = t\ p \\
fold\ (i,t,s,r,c)\ (Scale\ a) & = s\ a \\
fold\ (i,t,s,r,c)\ (Rotate\ a) & = r\ a \\
fold\ (i,t,s,r,c)\ (Compose\ (f,g)) & = c\ (fold\ (i,t,s,r,c)\ f, \\
 & \qquad fold\ (i,t,s,r,c)\ g)
\end{array}$$

Compositional interpretations can be expressed as folds; for example,

$$transform'\ t = fold\ (id, (+), scale, rotate, uncurry\ (\circ))\ t$$

One can see folds as the essence of compositional interpretations. And this gives us a clue about supporting multiple interpretations in a shallow embedding: if interpretations in a shallow embedding have to be compositional, and compositional interpretations are all and only those expressible as folds, then

> the fold pattern is precisely the least common generalization of all shallow interpretations.

The folding pattern is what all shallow interpretations have in common; the instantiation of the pattern—that is, the specific algebra—is what varies. Consider a version of *fold* with its arguments in the opposite order:

$$flip\ fold :: Transform \to (TAlg\ a \to a)$$

We should use the result type $TAlg\ a \to a$ of this function as the semantic domain for our *parametrized* shallow embedding; it can then be instantiated to any fold by supplying the corresponding algebra.

$$\mathbf{type}\ Transform_A = \forall a.\ TAlg\ a \to a$$

(For technical reasons, the type parameter a above has to be explicitly quantified rather than left unbound.) All the constructors of the language can be implemented easily under this representation:

$$\begin{array}{ll}
identity_A :: Transform_A & \\
identity_A\ (i,t,s,r,c) & = i \\[4pt]
translate_A :: Complex \to Transform_A & \\
translate_A\ p\ (i,t,s,r,c) & = t\ p \\[4pt]
scale_A :: Real \to Transform_A & \\
scale_A\ a\ (i,t,s,r,c) & = s\ a \\[4pt]
rotate_A :: Real \to Transform_A & \\
rotate_A\ a\ (i,t,s,r,c) & = r\ a \\[4pt]
compose_A :: (Transform_A, Transform_A) \to Transform_A & \\
compose_A\ (f,g)\ (i,t,s,r,c) = c\ (f\ (i,t,s,r,c), g\ (i,t,s,r,c)) &
\end{array}$$

And any compositional interpretation arises from applying the shallow embedding (which is a fold computation) to the appropriate algebra:

$$\begin{array}{l}
transform_A :: Transform_A \to (Complex \to Complex) \\
transform_A\ t = t\ (id, (+), scale, rotate, uncurry\ (\circ))
\end{array}$$

Many seemingly non-compositional interpretations are still expressible as folds, if looked at in the right way. For example, the interpretation *isLinear* above is non-compositional, because to determine whether $Compose\ (t,u)$ is linear, it does not suffice to know whether t and u are linear. Still, it is a simple projection from *transform*, which is compositional:

$$isLinear\ t = (transform\ (t,0) == 0)$$

Mutually dependent interpretations can be defined together as a pair. Context-dependent interpretations, such as precedence-aware printing, can be turned into context-independent compositional higher-order interpretations:

$$printPrec :: Transform \to (Precedence \to String)$$

5. Conclusion

Deep and shallow embeddings are more popular in functional programming circles than in object-oriented ones. That's not so surprising, give as we have seen that they depend heavily on algebraic datatypes and higher-order functions, respectively. Still, modern language design (C#, Scala, Python) combines the best of both paradigms, so hopefully that barrier will gradually recede. Then the lightweight embedded approach will become more widely available.

6. Acknowledgements

This talk proposal is based on a functional pearl [7] appearing at ICFP 2014, joint work with Nicolas Wu, which in turn grew out of lectures [6] given at the Central European Functional Programming summer school in 2013. The work was supported by EPSRC research grant EP/J010995/1 on *Unifying Theories of Generic Programming*.

References

[1] J. Bentley. Programming pearls: Little languages. *Comm. ACM*, 29(8):711–721, 1986. Also Chapter 9 of [2].

[2] J. Bentley. *More Programming Pearls: Confessions of a Coder*. Addison-Wesley, 1988.

[3] R. Boulton, A. Gordon, M. Gordon, J. Harrison, J. Herbert, and J. V. Tassel. Experience with embedding hardware description languages in HOL. In *Theorem Provers in Circuit Design*, pages 129–156. North-Holland/Elsevier, 1992.

[4] M. Erwig and E. Walkingshaw. Semantics-driven DSL design. In M. Mernik, editor, *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*, pages 56–80. IGI-Global, 2012.

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

[6] J. Gibbons. Functional programming for domain-specific languages. In V. Zsók, editor, *Central European Functional Programming Summer School*, volume 8606 of *LNCS*, pages 1–27. Springer, 2014. To appear.

[7] J. Gibbons and N. Wu. Folding domain-specific languages: Deep and shallow embeddings. In *ICFP*, Sept. 2014.

[8] P. Hudak. Building domain-specific embedded languages. *Comput. Surveys*, 28(4), 1996.

[9] G. Hutton. A tutorial on the universality and expressiveness of fold. *J. Funct. Prog.*, 9(4), 1999.

[10] S. M. Kamin and D. Hyatt. A special-purpose language for picture-drawing. In *Domain-Specific Languages*. Usenix, 1997.