Domain Specific Embedded Languages

Luke Storry

December 18, 2016

Introduction

Although most commonly used languages are General-Purpose, often a specific niche (or domain) is better served by a language specifically tailored for its requirements, a Domain-Specific Language. With specific tools and common assuptions hard-coded into the language itself, writing for that specific domain becomes much easier and more expressive, even though the language may not work outside that domain, or not even be Turing-complete.

Defining Domain-Specific Languages

Domain-Specific Languages themselves have existed for a lot longer than the term itself has, in the forms of ”task-specific” or ”database-orientated” languages. Sometimes, whether a certain language is domain-specific or not can often be quite ambiguous.

However, in his book [3], Fowler set clear boundaries on the definition of what a Domain Specific Language (DSL) is:

> Domain Sepcific Language (noun): a computer programming language of limited expressiveness focussed on a particular domain.

He later goes on to describe the key four elements that are required for a DSL:

1. A DSL must be a Computer Programming Language - readable by humans and executable by computers.
2. The nature of a DSL must be such that there is fluency, and difference expressions can be composed together.
3. Unlike a General-Purpose Language, a DSL must only support a bare minimum of features needed to support its domain.
4. A limited language is only useful if it has a clear focus on a limited domain

The two main ways of implementing DSLs are to either build a standalone language, which can be later compiled or translated to execute, or to embed it within another, previously-existing language. This latter option has the added benefit of keeping the already-familiar syntax of the host language, but only using a subset of its features

in the definition of the DSL. This also drastically reduces the initial expense of setting up the language - everything from compilers and syntax checking to debugging tools from the host language can be re-used by the embedded language.

A very functional and strongly-typed language, such as Haskell, with its higher-order functions, algebraic datatypes and lazy evaluation prove to be very well-suited for embedding.

Embedding

Embedding a domain-specific language involves writing a set of combinators in the host language that defines the syntax and semantics of the new DSL.[2]

Deep and Shallow Embedding

The embedding may be deep or shallow, depending on whether terms in the language construct syntactic or semantic representations. [5]

A shallow embedding only implements the terms from the DSL as values to be evaluated, but in a deep embedding the terms in the DSL construct an abstract syntax tree, which is traversed for evaluation.

Integer Addition Example

To demonstrate this distinction, embedding the simple language of integer addition with both types of embedding is shown below. This language will have only two constructors, *lit*, which will generate our Int type, and *add*, which will add two of them together.

```
type Int = ...
lit :: Integer -> Int
add :: Int -> Int -> Int
```

The expression $(1+(2+3))$ would be evaluated by: `eval(add(lit 1) (add(lit 2)(lit 3)))`

Shallow Embedding

To embed this language in Haskell in a shallow fashion, the language is defined directly in terms of its semantics:

```
type Int = Integer
lit n = n
add x y = x+y
eval :: Int  Integer
eval n = n
```

Deep Embedding

In a deep embedding however, the two operators are directly encoded in the host language as constructors of new datatype:

```
data Int ::  where
   Lit  :: Integer -> Int
   Add  :: Int -> Int -> Int
   eval :: Int -> Integer
lit n = Lit n
add x y = Add x y
```

This is a representation of the abstract syntax as an algebraic datatype, with functions to assign semantics by traversing the datatype.

Circuits Example

A more involved example is the construction of parallel prefix circuits. [5] This takes a sequence of inputs $x_1, x_2, ..., x_n$ and produces the sequence $x_1, x_1 \cdot x_2, \ldots, x_1 \cdot \ldots \cdot x_n$ as outputs. The operator $\cdot$ denotes a local operation that combines the two inputs.

Construction

These circuits can be constructed with the operators:

```
type Size = Int
type Circuit = ...
identity :: Size -> Circuit
fan :: Size -> Circuit
above ::Circuit -> Circuit -> Circuit
beside ::Circuit -> Circuit -> Circuit
stretch ::[Size] -> Circuit -> Circuit
```

For example:
`identity 3:` `fan 4:`

The `above` and `beside` constructors simply place the two given circuits above or beside one another, and the `stretch` constructor creates a identity circuit for each item in the list, and connects their last wires together according to the original circuit.

Embedding

Shallow Embedding

As Shallow Embedding has no actual data structures, we have to decide a metric to output, for example the width of the circuit. Then a circuit could be shallow-embedded

in Haskell with the following code:

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

Deep Embedding

In a Deep Embedding, Circuits can be defined with an algebraic datatype:

```
data Circuit ::  where
    Identity :: Size -> Circuit
    Fan      :: Size -> Circuit
    Above    :: Circuit -> Circuit -> Circuit
    Beside   :: Circuit -> Circuit -> Circuit
    Stretch  :: [Size] -> Circuit -> Circuit
```

This datatype can then be manipulated with simple functions, similar to the shallow embedding, for example width can be computed with:

```
type Width = Int
width :: Circuit     -> Width
width (Identity w)   = w
width (Fan w)        = w
width (Above x y)    = width x
width (Beside x y)   = width x + width y
width (Stretch ws x) = sum ws
```

Comparison

As can be seen, often the deep and shallow embeddings can appear similar, but the semantic differences result in large benefits and drawbacks for each approach: For example, if we wanted to add a new node to the circuit, with a shallow embedding we can just define a new function to add the new semantic, but in a deep embedding, a new datatype would be required, as well as updating all the definitions.

Conversely, to add a new interpretation of the circuit, such as whether a given circuit is ”connected”, the shallow embedding would need to be entirely re-written, whereas the already-defined datatype in the deep embedding means that little work is required.

So, both types of embedding can be cumbersome in different ways. Shallow embeddings having their language constructs mapped directly to their semantics, which leads to more succinct implementations, while Deeper embeddings' abstract syntax tree can be more complex and harder to expand, can allow for transforming expressions and compiling code from them[1].

References

[1] Combining deep and shallow embedding of domain-specific languages. *Comput. Lang. Syst. Struct.*, 44(PB):143–165, December 2015.

[2] Robert Atkey, Sam Lindley, and Jeremy Yallop. Unembedding domain-specific languages. In *Proceedings of the 2Nd ACM SIGPLAN Symposium on Haskell*, Haskell '09, pages 37–48, New York, NY, USA, 2009. ACM.

[3] Martin Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 1st edition, 2010.

[4] Jeremy Gibbons. Functional programming for domain-specific languages (lecture), July 2013.

[5] Jeremy Gibbons and Nicolas Wu. Folding domain-specific languages: Deep and shallow embeddings (functional pearl). In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 339–347, New York, NY, USA, 2014. ACM.

[6] P. Hudak. Modular domain specific languages and tools. In *Proceedings of the 5th International Conference on Software Reuse*, ICSR '98, pages 134–, Washington, DC, USA, 1998. IEEE Computer Society.

[7] Paul Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es), December 1996.

[8] P. J. Landin. The next 700 programming languages. *Commun. ACM*, 9(3):157–166, March 1966.

[9] Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *Proceedings of the 2Nd Conference on Domain-specific Languages*, DSL '99, pages 109–122, New York, NY, USA, 1999. ACM.

[10] Morten Rhiger. A foundation for embedded languages. *ACM Trans. Program. Lang. Syst.*, 25(3):291–315, May 2003.

[11] Nicolas Wu. Language engineering lectures, October 2016.