Comparing Shallow and Deep Embedding

Karim Allaouat
Ka15579@bristol.ac.uk

1. Introduction

As the name suggests, a domain specific language (DSL) is a computer language with the specific purpose across a single domain. This is contrary to a general-purpose language (GPL) which can be utilised across many domains however, lacks specialised features of a DSL. A common example of a DSL is HTML, the standard markup language for creating web pages. A less common example is Emacs Lisp, a variation of the functional language Lisp specifically used as a scripting language by the GNU Emacs text editor. Originally comparing GPLs and DSLs, one can see that a DSL has advantages such as allowing domain experts to fully understand the language, however, there is a cost to teaching, creating and maintaining a DSL. It may seem that a DSL can be a waste of time to engineer (as GPLs are Turing complete) nevertheless, this generality means that it may take a great deal of programming to produce something that a DSL could do in a few lines as well as having fewer options of implementation.

There are two standard approaches to implementing a domain specific language. A standalone DSL comes with its own syntax and semantics, compiler and can be created to be maximally suitable for a task. This contrasts with embedding a DSL within a GPL (EDSL), that uses the compiler from the host GPL and is a series of definitions written in the GPL. An example of an EDSL is the Orc package that provides an EDSL with Orc primitives, with Haskell as the host language. Initially comparing embedded DSLs and standalone DSLs, it becomes apparent that it will be easier to implement an EDSL as it uses the GPLs development environment although, a standalone DSL can be designed for maximum convenience to the user.

Within embedding a DSL there are also two approaches. We can take the deep embedding (D-EDSL) approach by creating data types expressed in a syntax tree then optimised and transformed for evaluation. A shallow embedded DSL (S-EDSL) captures the semantics of the data of a domain within operations. Primarily relating shallow and deep, one can see that if a new property of the language is to be included then shallow embedding makes this easier as a new function is to be created. Contrastingly, this becomes more challenging in deep embedding.

This paper has the primary purpose of being a source of revision as well as exploring the connections between deep and shallow embedding. This means that the target audience will be those who interested in understanding differences between types of embedding in a DSL. This paper will:

- Explain the nature of embedding languages.
- Understand and expand the examples from the lectures.
- Compare deep and shallow embedding such as the ease of adding new language constructs and interpretations.
- Create a new example embedded in Haskell using both implementations.
- Compare the runtimes of both embedding techniques to see if there is a difference in performance.

The motivation to fully understand the differences between shallow and deep embedding in a DSL is productivity. A useful DSL must be easy to utilise and understand in addition to concealing features outside the domain. Furthermore, it can significantly change how the language is created and used.

2. Initial Comparison

A paper by Josef Svenningsson of the Chalmers University of Technology titled 'Combining Deep and Shallow Embedding for EDSL', briefly compares the pros and cons of both embedding styles. He describes the pros of shallow embedding as:

- Easy to add new language constructs.
- Efficient, tagless evaluation.

And a con being that it doesn't allow for adding a new interpretation. In terms of deep embedding, the pros are:

- Easy to add new forms of interpretations.
- Possible to optimise representation.

However, the cons are that:

- It is arduous to add new language constructs.
- Potentially slow evaluation.
- More code is used.

It will be investigated as to what extent these points are true and increase understanding.

Looking at the example from the lectures, a circuit is a series of lines that can be connected and an operation performed upon them. It can be seen as a generic representation of an electronic circuit.

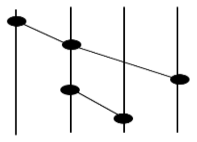


Figure 1. Example of a basic circuit.

As a deeply embedded DSL, a circuit is defined by data constructors that relate to its width.

```
data Circuit = Identity Int
             | Above Circuit Circjuit
             | Beside Circuit Circuit
```

The Identity constructor relates to the basic form of a circuit. Above portrays a circuit of width n placed above another circuit of width n. The Beside constructor conveys the idea of placing two circuits next to each other.

Contrastingly, as a shallowly embedded DSL, language constructs are expressed as functions.

```
identity w = w
above w1 w2 = w1
beside w1 w2 = w1 + w2
```

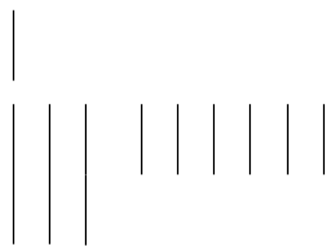


Figure 2. Example of Identity 1, Above (Identity 3) (Identity 3) and Beside (Identity 2) (Identity 4)

If another construct is to be added to our definition of a circuit, this can be done quite simply in the shallow DSL – a new function is to be added.

```
fan w = w
stretch ws w = sum ws
```

Comparing this to a deeply embedded DSL, a new data constructor must be created and every evaluative function altered to evaluate the new data.

```
Data Circuit = …
             | Fan Int
             | Stretch [Int] Circuit
```

A Fan circuit connects the first line of the circuit to the others. Stretch represents a list of non-empty positive widths of length n and a circuit of width n as well as some interconnecting lines at specific points.

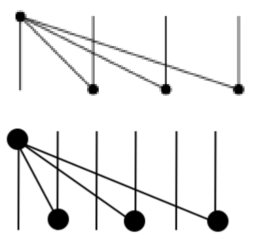


Figure 3. Example of Fan 4 and Stretch [1,2,2,1] (Fan 4)

The shallow embedded DSL evaluates the width of a circuit by default, whereas in the deeply embedded DSL, the width function is used.

```
width :: Circuit -> Int
width (Identity w)    = w
width (Above c1 c2)   = width c1
width (Beside c1 c2) = width c1 +
                        width c2
width (Fan w)         = w
width (Stretch ws c) = sum ws
```

If another interpretation of a circuit is to be added, then deep embedding seems to prevail over shallow embedding. A circuit is now to be defined as its depth. A new function can simply be added to the D-EDSL.

```
depth :: Circuit -> Int
depth (Identity d)    = 1
depth (Above c1 c2)   = depth c1 +
                        depth c2
depth (Beside c1 c2) = max (depth c1)
                        (depth c2)
depth (Fan d)         = 1
depth (Stretch ws c) = depth c
```

In the S-EDSL, instead of returning the width, every function is changed to return the depth. A more appropriate alteration would be to make the functions take in and return a tuple in the form (width, depth).

```
identity w              = (w, 1)
above (w1,d1) (w2,d2)   = (w1,d1+d2)
beside (w1,d1) (w2,d2) =
              ((w1+w2), max d1 d2)
fan w                   = (w, 1)
stretch ws (w,d)        = (sum ws, d)
```

Further analysis of a circuit shows that it must be well connected. The S-EDSL will now also return a Boolean value that represents if it is well connected – (connected, width, depth). This may get confusing and laborious to keep writing, therefore it can be defined as a circuit.

```
type Circuit = (Bool, Int, Int)

identity :: Int -> Circuit
identity w = (True, w, 1)
```

```
above :: Circuit -> Circuit -> Circuit
above (c1,w1,d1) (c2,w2,d2) = ((c1 &&
c2 && w1 == w2), w1, d1 + d2)

beside :: Circuit -> Circuit ->
Circuit
beside (c1,w1,d1) (c2,w2,d2) = ((c1 &&
c2 && d1 == d2), w1 + w2, d1)

fan :: Int -> Circuit
fan w = (True, w, 1)

stretch :: [Int] -> Circuit -> Circuit
stretch ws (c1,w,d) = ((c1 && length
ws == w), sum ws, d)
```

In the deeply embedded language, a new evaluative function is added.

```
connected :: Circuit -> Bool
connected (Identity d)    = True
connected (Above c1 c2)   = (connected
       c1) && (connected c2) && (width
       c1 == width c2)
connected (Beside c1 c2) = (connected
       c1) && (connected c2)
connected (Fan d)          = True
connected (Stretch ws c) = (connected
       c) && (length ws == width c)
```

These newly added features clearly show the difference in the difficulty of adding new language constructs and interpretations within a DSL. In conclusion, it is much easier to add new language constructs in shallow embedded languages however, it is challenging to change the interpretation once it is made. Moreover, the strengths of using deep embedding techniques are that it is easy to add new forms of interpretations, although, it is laborious to add new language constructs. Another point to make is that deep embedding uses more code as it requires a new data type for the abstract syntax tree.

3. A New Example

Within a computer, a series of logic gates and transistors can be abstracted into functioning modules. Simple processors and solutions can be made from these modules. A basic problem to solve is that of traffic lights. At each iteration, traffic lights need to change from red, to red and amber, to green, to amber, back to red again. This can be represented in 3 bits; 0 signifies off, 1 signifies on. The least significant bit represents the state of the red light; the next bit portrays amber and the 3rd bit relates to the green light. A single traffic light cycle is therefore:

0001	(Red)
0011	(Red and Amber)
0100	(Green)
0010	(Amber)

In terms of simple modules, this problem can be solved by having a multiplexer with these inputs (a) and a counter cycling between them (b).

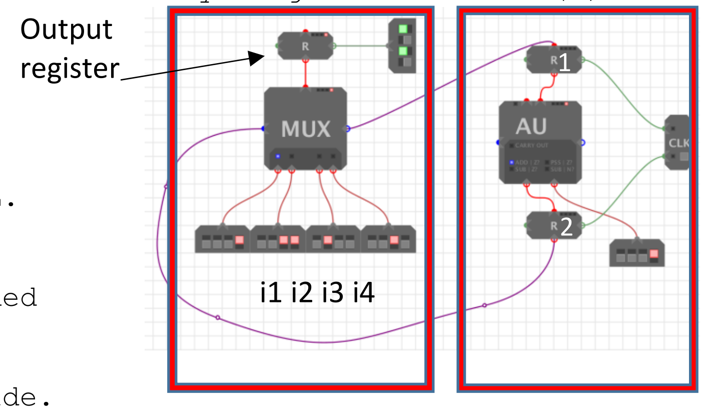


Figure 4. (a) in the box on the left joined with (b), on the right.

The clock will tick then the counter will increment and every four increments, the multiplexer will change the output value.

The language of functioning modules is in itself, a DSL. This language is to be embedded in Haskell by both deep and shallow methods then its performance compared over a number of iterations. A Module is defined as a:

- Switch
- Logic Unit
- Arithmetic Unit
- Register
- Multiplexer
- Clock

A switch contains an integer value and can be seen as the identity constructor of a module. Logic and arithmetic units take inputs from three modules in the form: (control, input1, input2). Control inputs decide which logical/arithmetic operation is performed upon the two inputs. A register receives a control input and a variable input as well as having a 4-bit memory. The multiplexor takes a control module followed by four modules used for input. The clock module takes an input that represents which output is being used.

In Module_Deep.hs, this is represented as a data type.

```
data Module = Switch Int
    | LogicUnit Module Module Module
    | ArithmeticUnit Module Module
Module
    | Register Module Module Int
    | Multiplexer Module Module Module
Module Module
    | Clock Int
```

Subsequently, an evaluation function is used to deduce the state of the module according to the rules that make up each module. The module representing the traffic light circuit can then be defined after time n.

```
trafficLights :: Int -> Module
trafficLights n = do
  let i1 = Switch 1
      i2 = Switch 3
      i3 = Switch 4
      i4 = Switch 2
      i5 = Switch 1
      i6 = Switch 0
      r1 = Register (Clock 0) au n
      r2 = Register (Clock 1) r1 n
      au = ArithmeticUnit i5 r2 i6
      mux = Multiplexer r1 i1 i2 i3 i4
      in mux
```

For example, at the fourth iteration, 'evaluate trafficLights 3' evaluates as 0010 (amber) which is as expected.

In Module_Shallow.hs, modules are represented as functions that are of type that the corresponding module.

```
switch :: Int -> Int
logicUnit :: Int -> Int -> Int -> Int
arithmeticUnit :: Int -> Int -> Int ->
Int
register :: Int -> Int -> Int -> Int
multiplexer :: Int -> Int -> Int ->
Int -> Int -> Int
clock :: Int -> Int
```

The definition of the traffic light circuit is also a function.

```
trafficLights :: Int -> Int
trafficLights n = do
      let i1 = switch 1
      i2 = switch 3
      i3 = switch 4
      i4 = switch 2
      i5 = switch 1
      i6 = switch 0
      r1 = register (clock 0) au n
      r2 = register (clock 1) r1 n
      au = arithmeticUnit i5 r2 i6
      mux = multiplexer r1 i1 i2 i3 i4
      in mux
```

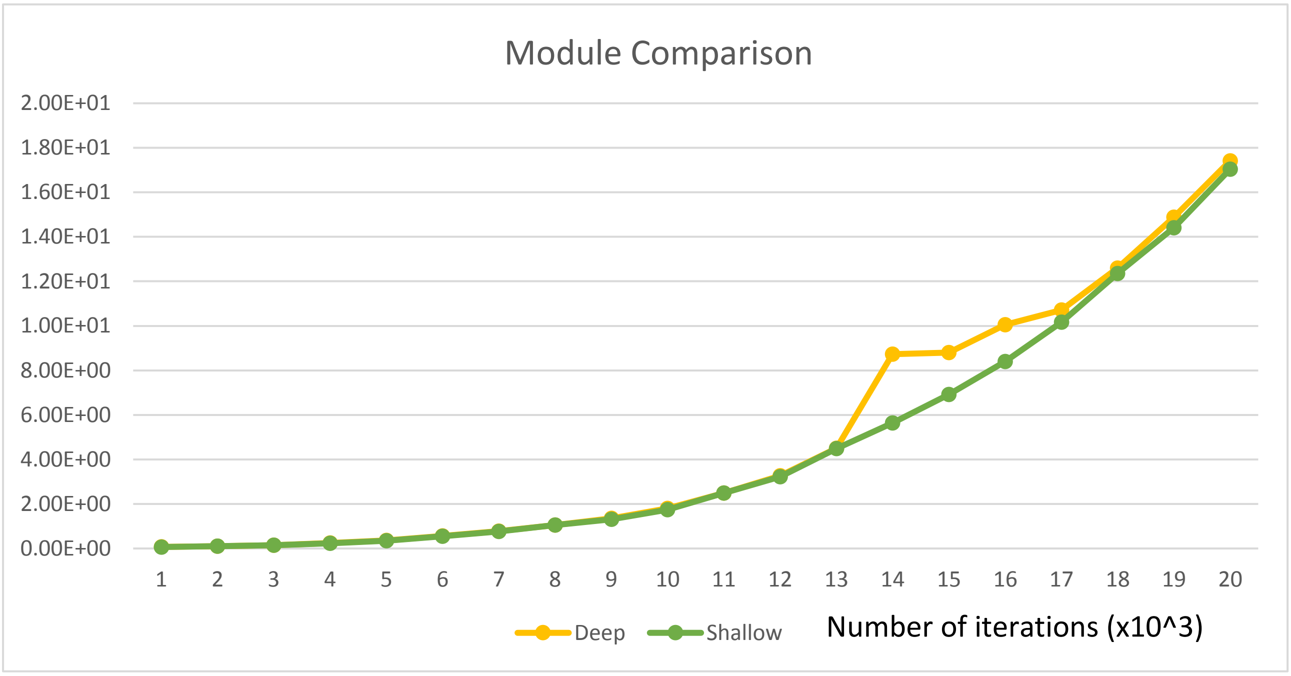
For instance, 'trafficLights 3' evaluates to 0010 which is the same as the deep example.

From a naïve analysis of D-EDSLs and S-EDSLs, it seems that shallow embedding would be faster. This is due to the fact that deep embedding requires the construction and traversal of an interim data structure that represents the expression. Having said this, the data structure can also be optimised and transformed to save time and space. It is then interesting and important in understanding, to compare the runtime of this example as n gets larger.

4. Evaluation Comparison

A timer function was added to measure the runtime of pure functions that directly outputted the results to a text file. These results were repeated ten times, an average found then extracted and a graph created.

These results imply that, as predicted, shallow embedding executes faster. This is due to the fact that an intermediate data structure does not need to be produced and modified. On the contrary, if the example was that of a more simple data structure or the interpretation of was complex, then deep embedding may be faster.



The shallow embedding example executes faster than the deep embedding example at every interval, although not by much. Due to the shape of the graph, it looks like both techniques are bounded by O(a^x). The shallowly embedded example makes a steady curve – as does the deeply embedded example, however contains an anomaly. Around iteration 14000, the average time for D-EDLS would increase by more than a second compared to what was expected. This jump is even more peculiar when the fact that it is repeatable. As to what causes this sudden jump, it opens a door for the opportunity of further research.

In conclusion, and to agree with Svenningsson, it is better to use shallow embedding when only a single, tagless interpretation is needed and when prototyping code. On the other hand, deep embedding is useful when compiling to a language and when multiple interpretations are required. Lindsey Kuper, a parallel computing research scientist at Intel Labs writes, "In fact, I see people concluding that deep embeddings are *better*, which seems to me about as nonsensical as saying that high-level languages are better than low-level languages." Therefore, it is best to analyse which technique is best fitting for the task at hand.

In the future, it will be useful to research where these embedding techniques can be used together to make an optimal solution and have a nice interface.

5. Additional Code

```
module Module_Deep where

import Data.Bits
import System.Environment
import Timer

data Module = Switch Int
             | LogicUnit Module Module Module
             | ArithmeticUnit Module Module Module
             | Register Module Module Int
             | Multiplexer Module Module Module Module Module
             | Clock Int
             deriving Show

-- Evaluate a module after n steps.
evaluate :: Module -> Int -> Int
evaluate (Switch i) n = mod i 16
evaluate (LogicUnit c i1 i2) n
               | mod (evaluate c n) 4 == 0 = (15 - (mod (evaluate i1 n) 16))
               | mod (evaluate c n) 4 == 1 = ((evaluate i1 n) .&. (evaluate i2 n))
               | mod (evaluate c n) 4 == 2 = ((evaluate i1 n) .|. (evaluate i2 n))
               | mod (evaluate c n) 4 == 3 = (xor (evaluate i1 n) (evaluate i2 n))
evaluate (ArithmeticUnit c i1 i2) n
               | mod (evaluate c n) 4 == 0 = ((evaluate i1 n) + (evaluate i2 n))
               | mod (evaluate c n) 4 == 1 = (evaluate i1 n)
               | mod (evaluate c n) 4 == 2 = ((evaluate i1 n) - (evaluate i2 n))
               | mod (evaluate c n) 4 == 3 = ((evaluate i1 n) - (evaluate i2 n))
evaluate (Register c i mem) n
               | (evaluate c n) == 0 = mod mem 16
               | (evaluate c n) == 1 = (evaluate i n)
               | otherwise           = 0
evaluate (Multiplexer c i1 i2 i3 i4) n
               | mod (evaluate c n) 4 == 0 = (evaluate i1 n)
               | mod (evaluate c n) 4 == 1 = (evaluate i2 n)
               | mod (evaluate c n) 4 == 2 = (evaluate i3 n)
               | mod (evaluate c n) 4 == 3 = (evaluate i4 n)
evaluate (Clock t) n
               | mod t 2 == 0 = mod n 2
               | otherwise    = 1 - (mod n 2)

trafficLights :: Int -> Module
trafficLights n = do
            let i1 = Switch 1
                i2 = Switch 3
                i3 = Switch 4
                i4 = Switch 2
                i5 = Switch 1
                i6 = Switch 0
                r1 = Register (Clock 0) au n
                r2 = Register (Clock 1) r1 n
                au = ArithmeticUnit i5 r2 i6
                mux = Multiplexer r1 i1 i2 i3 i4
                in mux

module Module_Shallow where

import Data.Bits
import System.Environment
import Timer

switch :: Int -> Int
switch i = i

logicUnit :: Int -> Int -> Int -> Int
logicUnit c i1 i2
    | mod c 4 == 0 = (15 - (mod i1 16))
    | mod c 4 == 1 = (i1 .&. i2)
    | mod c 4 == 2 = (i1 .|. i2)
    | mod c 4 == 3 = (xor i1 i2)

arithmeticUnit :: Int -> Int -> Int -> Int
arithmeticUnit c i1 i2
    | mod c 4 == 0 = i1 + i2
    | mod c 4 == 1 = i1
    | mod c 4 == 2 = i1 - i2
    | mod c 4 == 3 = i1 - i2

register :: Int -> Int -> Int -> Int
register c i mem
    | c == 0    = mod mem 16
    | c == 1    = i
    | otherwise = 0

multiplexer :: Int -> Int -> Int -> Int -> Int -> Int
multiplexer c i1 i2 i3 i4
    | mod c 4 == 0 = i1
    | mod c 4 == 1 = i2
    | mod c 4 == 2 = i3
    | mod c 4 == 3 = i4

clock :: Int -> Int
clock t
    | mod t 2 == 0 = 0
    | otherwise    = 1

trafficLights :: Int -> Int
trafficLights n = do
            let i1 = switch 1
                i2 = switch 3
                i3 = switch 4
                i4 = switch 2
                i5 = switch 1
                i6 = switch 0
                r1 = register (clock 0) au n
                r2 = register (clock 1) r1 n
                au = arithmeticUnit i5 r2 i6
                mux = multiplexer r1 i1 i2 i3 i4
                in mux

module Timer where
-- Code from https://wiki.haskell.org/Timing_computations
import Text.Printf
import Control.Exception
import System.CPUTime
import Control.Parallel.Strategies
import Control.Monad
import Control.DeepSeq
import System.Environment

lim :: Int
lim = 10^6

time :: (Num a, NFData a) => a -> String -> IO ()
time y f = do
    start <- getCPUTime
```

```
replicateM_ lim $ do
    x <- evaluate (1 + y)
    rnf x `seq` return ()
end   <- getCPUTime
let diff = (fromIntegral (end - start)) / (10^12)
appendFile f (show (diff :: Double) ++ "\n")
return ()
```