Constructing And Improving Embedded Domain Specific Languages

Callum Pearce, December 17th 2016

1 Introduction

A DSL is a programming language created to serve a specialised purpose for a certain domain, they are often a subset of a general purpose language. General purpose languages on the other hand, are applicable across many domains [1]. DSLs can be used instead of general purpose languages to solve a given problem, as they are created to solve the problem itself and therefor can express the solution with higher clarity.

There are two types of DSLs, internal/embedded and external. Internal DSLs contain specific features for a specific domain, they are written in the host general purpose language, for example Rails in Ruby. External DSLs are parsed separately from the host general purpose language, for example, regular expressions [7]. This report will be focused upon two different types of embedded DSLs and how they are implemented. The languages will then be improved in a way which makes the solution clearer. The following points of discussion are included:

- Constructing a DSL and explaining how a DSL works.
- Deeply and shallowly embedded DSLs, including a example and analysis of both.
- Constructing an improved deep embedding, by moving recursion.
- Constructing an improved shallow embedding, by adding multiple semantic meanings to a single DSL.
- Comparing the improved deep and shallow embeddings.

2 Constructing a DSL

A Haskell example is a good way to show how to construct DSLs, due to the ease of defining new complex algebraic data structures. In the example internal DSL below a Block domain is covered. Blocks are essentially building block structures which can be constructed using the datatype definition. There are also a few functions which can return information about Blocks, see Figure 1 (extended) for the other semantics implemented. This simple example should clearly illustrate what a embedded DSL encompasses and how it is constructed.

Figure 1

```
> data Block =  Brick Int Int          --height, weight
>             | Combine Block Block    --Block1, Block2
>             | Beside Block Block     --Block1, BLock2
>             | Build Int Int Block    --height, weight, Block

--Returns the width of a block

> width :: Block -> Int
> width (Brick x w) = 1
> width (Combine b1 b2) = width(b1)
> width (Beside b1 b2) = width(b1) + width(b2)
> width (Build x w b) = width(b)

...
```

2.1 Figure 1 Analysis

The embedded DSL in Figure 1 has two main components, the data type and the functions which implement the semantics of the DSL. The data type consists of the four different value constructors which can be used to construct a block. Brick is simply the base case of a Block, all Blocks are structurally made of Bricks of a given height and weight. The three other value constructors build on top of this. Combine places two Blocks on top of one another. Beside places two given Blocks next to each other. Finally build creates a layer of bricks at the bottom of the Block of a given identical height and weight.

The functions below the AST define how a given semantic is implemented by giving a definition of how to correctly apply the function when any given value constructor for a Block is encountered. For example when width is called on a given Block and a Beside value constructor is encountered, the width of the new Block formed is equal to the sum of the widths of the Blocks given in the Beside value constructor. All functions within the DSL are defined in a similar way to width making the functionality of the DSL easy to understand. A description of the other implemented semantics for the Block domain is located in the Appendices section.

3 Deep And Shallow Embedding

Internal DSLs can either be deeply embedded or shallowly embedded. Deeply embedded DSLs are created in such a way that they have a newly made abstract syntax tree (AST), which is optimised and transformed in the process [1][2]. Shallowly embedded DSLs on the other hand are implemented by creating the semantic meanings directly using functions, rather than a data type[2]. This avoids the need for creating an AST, instead the pre existing languages ASTs are used in the computation of solutions. Below these concepts have been explained in terms of Haskell syntax to give a more in depth understanding.
In deeply embedded DSLs the data type of the DSL describes the domain covered by the DSL. Functions are defined in a way that they describe what is done for each value constructor, for the specific datatype. On the other hand shallowly embedded DSLs use functions to define features and the domain covered by the DSL and skip the need for a new AST to be created[1].

From this description of what the difference is between a deeply and shallowly embedded DSL, it is clear that Figure 1 represents a deeply embedded DSL. The in depth datatype definition states what value constructors the functions within the DSL need to provide a definition for. For the code in Figure 1, the AST is the data type definition for the Block data type. The functions below the data type implement the semantic meanings, using the value constructors provided by the data type.

Figures 2.(1-4) shows 4 shallowly embedded DSLs for the Block DSL, together they cover the same domain as the Figure 1 code and have the same functionality.

Figure 2.1

Shallow Embedding for the width Semantic

```
> type Block = Int

> brick :: Int -> Int -> Int
> brick h w =  1

> combine :: Block -> Block -> Int
> combine b1 b2 = b1

> beside :: Block -> Block -> Int
> beside b1 b2 = b1 + b2

> build :: Int -> Int -> Block -> Int
> build x w b = b
```

3.1 Figure 2.1 Analysis

Figure 2.(1-4) represents multiple shallow embeddings which together make the same DSL as that displayed in Figure 1, Figure 2.1 represents a single DSL. This DSL can calculate the width of a given Block correctly. This is the only semantic implemented, as with this way of defining a DSL it is not possible to define Multiple instances of brick, combine, beside and build functions in a single language without the use of type classes (this will be implemented later). Essentially the functions create the Block structure as a single integer which is the width. Figures 2.(2-4) define other semantic meanings, each in a standalone language for this reason.

3.2 Comparing deep and shallow embeddings

After analysing Figures 2.(1-4) and Figure 1 the two following conclusions can be drawn:

Shallowly embedded DSLs avoid using a newly created AST, this makes their evaluations quicker. However the lack off an AST means they are not equipped to deal with multiple interpretations (multiple semantics) within the DSL. This is why in Figure 2.(1-4) there are four different shallow DSLs written to have the same functionality as the single deep embedding. Essentially if a new semantic meaning needs to be added to the current shallow embedding, the whole DSL would have to be re-written to implement this. On the other hand, the deeply embedded DSL allows us to have many different interpretations all in the same DSL.

The deeply embedded DSL uses recursion to define the different semantics of the DSL. This is not ideal as it is not as clear how the final value was obtained from the function, compared to an implementation which avoids the use of recursion. The shallow embedding however, does not use recursion in its implementation for a given semantic. This makes it easier to understand how the solution is acquired.

4 Moving recursion from the deep embedding

As discussed above the deeply embedded DSL in Figure 2 uses recursion to define the semantics of the language. This can be fixed in a way by moving the recursion from the function definitions to another function which handles the recursion. The idea is to create a generic function which handles recursion within the language[5]. The code below shows how such a concept can be implemented.

Figure 3

```
> data Block k = Brick Int Int
>              | Combine k k
>              | Beside k k
>              | Build Int Int k

> data Fix f = In(f (Fix f))

> inop :: Fix f -> f (Fix f)
> inop (In x) = x

> cata :: Functor f => (f a -> a) -> Fix f -> a
> cata alg = alg . fmap (cata alg) . inop

> instance Functor Block where
>   fmap :: (a -> b) -> Block a -> Block b
>   fmap f (Brick h w) = Brick h w
>   fmap f (Combine k1 k2) = Combine (f k1) (f k2)
>   fmap f (Beside  k1 k2) = Beside (f k1) (f k2)
>   fmap f (Build h w k) = Build h w (f k)

> width :: Fix Block -> Int
> width = cata alg where
>   alg :: Block Int -> Int
>   alg (Brick h w) = 1
>   alg (Combine b1 b2) =  b1
>   alg (Beside b1 b2) = b1 + b2
>   alg (Build h w b) = b

...
```

4.1 Data Types

Comparing the data type definition between Figure 1 and 3, the following observations can be made. Figure 3 includes a Block k on the left side of the definition whilst Figure 1 does not. Also where there is a Block written on the right hand side of the definition in Figure 1 it is instead replaced with a k in Figure 3. These are the only differences between the new and improved data type and the old one. The k can be thought of as the ‘k’ontinuation of Blocks to follow, we use recursion in the definition of the data type in Figure 1 where as in Figure 3 we use k to represent recursions location rather than directly defining it. Essentially the data type gives the structure of a Block and marks the recursive areas. The Fix data type is what actually handles the recursion of the data type.

The Fix data type is used to represent the recursive Block structure. A Block can be made of many Blocks this makes a Block recursive in it's own definition, the Fix data creates a definition which supports this by giving a Block made of one or more Blocks a type Fix Block. The Fix data type allows recursive functions to be represented in a non-recursive way [8]. This allows the recursive structure of a Block to be passed in to functions such as width because Fix Block means a Block made of Blocks. But how does the function then manipulate this Fix Block structure, there is no way to tell how many Blocks are in the structure and of what their attributes are? This is exactly what the inop function solves, by deconstructing the Fix Block type. This is shown by the type definition of inop. There is a short expansion of the recursive Fix data type below.

```
f -> f (Fix f) -> f (f (Fix f)) -> f(f (f (Fix f))) -> ....
```

4.2 Block Functor

A Functor is simply a data type which has a definition of the function fmap for each of its value constructors. This means that if a data type has a functor instance than it is possible to unwrap the value out of it's context within the data type, and apply a function correctly to the value, then wrap the new value in it's previous context. Functors are only valid if the following laws hold true[4]:

```
fmap id = id                               --(1st Law)
fmap (e.g) = (fmap e) . (fmap g)           --(2nd Law)
```

The first law states that mapping the identity function over any part of the data structure has no affect. This is true for the Block functor instance as if the function f simply returned the value it was applied to, there would be no change to the total Block's structure. The structure of the original Block remains the same after any function is applied to the Block data structure, therefor the second rule holds true. Below there are two proofs, one for each rule. They are both for the Combine fmap definition, the proof for Beside and Build are almost identical. As for Brick the function f is not applied to any values therefor its proof is trivial[4].

Figure 4

 In Figure 3 the Functor instance definition describes how to apply a function to any of the value constructors (contexts) of a Block. The data type of fmap should consolidate this concept as a function and a Block are taken as input, then a new Block is produced after. This Block produced has had the function applied correctly to the data within the Block's context. The Block Functor instance points to the recursive site within the Blocks value constructors. This is required to map the cata function in to a Block's value constructors elements which contains another Block structure, this allows cata to be applied to the entire recursive structure of a Block.

4.3 Cata

Now the Block Data type and Functor instance are in place, cata uses them to apply a given function to a given Block by replacing all of the In's in the Block structure with the algebra function passed in (A full description of how this works is given in the Functions section). The cata function essentially does this by applying a given algebra to the nested Block data structure. On a type level cata takes a function which converts a functor f (a Block type) with value a (f a) to type a, which is the passed in algebra. A Fix f variable is also taken in (Fix Block) which is the Block the given algebra is applied to. The following diagram illustrates what cata does on a type level.

```
fmap id (Combine k1 k2) == Combine k1 k2
{-Proof: First law of Functors-}

fmap id (Combine k1 k2)
=
Combine id(k1) id(k2)
=
Combine k1 k2                --Therefor first law is prooved true.
```

```
fmap (g.f) (Combine k1 k2) == (fmap g . fmap f) (Combine k1 k2)
{-Proof: Second law of Functors-}

= {- Proove by applying fmap on both sides of the equality in the second Law, start with left hand side-}
Combine (g.f)k1 (g.f)k2
=
Combine g(f k1) g(f k2)

= {- Now attempt to equal right hand side to what the left hand side has been simplified to. -}
fmap g (fmap f (Combine k1 k2))
=
fmap g (Combine (f k1) (f k2))
=
Combine g(f k1) g(f k2)     -- right and left sides simpliify to the same value, therefor true.
```

Figure 5

```
      fmap(cata alg)

f (Fix f) -------> f a
    /\              |
    |               | alg
inop|               |
    |               \/
 Fix f     ------->  a
        cata alg
```

The diagram above illustrates the components of cata alg and how they combine together to apply a given algebra to a recursive data structure. For the DSL in Figure 3 the story begins at type Fix f, which is the data structure of a Block wrapped in In's. The inop deconstructs this and transforms this for they current layer of recursion in the data structure to produce type f (Fix f) [5]. Next fmap(cata alg) is applied which recursively applies cata alg to the entire data structure, deconstructing all In's and applying the algebra to all value constructors in the Block data structure. This changes each layer of the recursive Block structure to have type f a, where f is the algebra to be applied and a is the value constructor with its stored value(s). The algebra is then applied to every value constructor and it's contents starting from the recursive base case and working up the data structure until a final single value is produced. This produced value is the solution to applying a given semantic to the entire recursive data structure, returning type a (the return type specified by the algebra).

4.4 Functions

Function Types

```
alg :: Block Int -> Int
```

At every layer of recursion alg is applied converting the Block structure to an int which represents the current value at that point in the recursive Block data structure. However the Block structure is represented as an Int, which is what

Block Int means. The Int produced by alg is used in the next level up of recursion as input, this continues until the top of the recursive Block data structure is met. The value produced by the function at this point is the solution.

```
'function' :: Fix Block -> Int
```

The 'function' e.g. width calls alg on every layer of the recursive structure of a Block using the cata function (a full description is shown below for this step). This produces a single integer which is the calculated width.

Function Definitions

As explained above the cata function applies a specific algebra to every layer of recursion in the Block structure, therefor in every semantics function implementation, a specific algebra for the function is defined. Every functions alg definitions state what the value produced by the function is when a specific value constructors content is met.

The algorithm for applying a function in the language (Example displayed in Figure 6)

1. Expand the cata function using it's definition.
2. Apply functions from right to left to the value constructors and their contents which make up the Block. This will begin with inop.
3. Apply fmap(cata alg) using the definition specified for the value constructor it is being applied to, located in the instance of fmap definition.
4. Repeat steps 1-3 until there are no more inop or fmap(cata alg) functions to apply.
5. Apply alg function by using the definition for the specific algebra for the input. Start from inner most instances of alg's i.e. the base cases in this case it would be applying the alg to innermost Brick's first. Once all alg's have been applied then the function has been applied and the correct value is produced.

Figure 6

```
    width(In(Beside (In(Brick 4 4)) (In(Brick 4 4))))

 == {- width = cata alg -}
    (cata alg) (In(Beside (In(Brick 4 4)) (In(Brick 4 4))))
 == {- Expand cata function -}
    (alg . fmap (cata alg) . inop) ( In(Beside ( In(Brick 4 4) ) ( In(Brick 4 4) )))
 == {- Apply from left to right, apply inop -}
    (alg . fmap (cata alg)) (Beside ( In(Brick 4 4) ) (In(Brick 4 4) ))
 == {- Apply fmap (cata alg), to do this use fmap definition for Beside  (the recursive step) -}
    (alg . (Beside ((cata alg) ( In(Brick 4 4) ))  ((cata alg)  ( In(Brick 4 4) ))))
 == {- Expand left cata function -}
    (alg . (Beside ((alg . fmap(cata alg) . inop) ( In(Brick 4 4) )) ((cata alg) ( In(Brick 4 4) ))))
 == {- Apply inop -}
    (alg . (Beside ((alg . fmap(cata alg)) (Brick 4 4)) ((cata alg) ( In(Brick 4 4) ))))
 == {- Apply fmap(cata alg) using fmap definition for Brick -}
    (alg . (Beside (alg . (Brick 4 4)) ((cata alg) ( In(Brick 4 4) ))))
 == {- Expand cata function -}
    (alg . (Beside (alg . (Brick 4 4)) ((alg . fmap(cata alg) . inop) ( In(Brick 4 4) ))))
 == {- Apply inop -}
    (alg . (Beside (alg . (Brick 4 4)) ((alg . fmap(cata alg)) (Brick 4 4))))
 == {- Apply fmap(cata alg) -}
    (alg . (Beside (alg . (Brick 4 4)) (alg . (Brick 4 4))))
 == {- Now apply the alg function defined for width for each value constructor -}
    (alg . Beside (alg . (Brick 4 4) (1)))
    (alg . (Beside (1) (1)))
    (2)
```

5 Adding Multiple Semantics to the Shallow Embedding

Now it is time to cover how the shallow embedding for the Block language can be improved. The shallow embedding in Figure 2.1 had the advantage of not using recursion therefor this does not need to be solved, unlike the deep embedding. However the current shallow embedding only supports one semantic meaning per DSL, so Figure 2.(1-4) displays four different DSLs. In this section a single improved shallow embedding is created which supports all the semantics which are covered by the deeply embedded DSL, this is achieved by using typeclasses. Figure 7 displays the improved shallow embedding and a breakdown on it's implementation is below the Figure.

Figure 7

```
> type BlockWidth = Int

> class Block block where
>   brick   :: Int -> Int -> block
>   combine :: block -> block -> block
>   beside  :: block -> block -> block
>   build   :: Int -> Int -> BlockWidth -> block -> block

> newtype Width = Width{width::Int}

> instance Block Width where
>   brick h w         = Width 1
>   combine b1 b2     = b1
>   beside b1 b2      = Width(width b1 + width b2)
>   build h w width b  = b

...
```

5.1 Typeclass

Typeclasses are used to define some behaviour and then types which can behave in the way defined can have their own instance of the typeclass. For example for a type to be an instance of the Eq typeclass it needs to have within the instance declaration a definition of the (==) operation. Not all types are part of the Eq typeclass and rightly so, only ones which can behave in a way where there equality found within the type. An example of an instance of the Eq typeclass is the Int instance, this makes sense as two Ints can equal one another, same goes for Chars, Strings and many more [6]. The Block typeclass defines the behaviour of a Block, which includes placing bricks, combining two Blocks together, placing two Blocks next to each other and building a block up. The type instances of the typeclass are created for types which exhibit the Block behaviour. For example the Width type created is made to represent the Width of a given Block, therefor it exhibits the behaviour of the Block typeclass and an instance can be made.

In the Block language case the typeclass for a Block defines what functions must be implemented in the interface as well as the type signatures of the functions. This is used to create a shallow embedding which can support all of the semantics covered by the deeply embedded DSL. For every semantic there is a new type and instance of the Block type class which implements the semantic. The block type in the type signatures represents the type which will be used to create an instance of the Block typeclass. For example this could be Width, Height, Weight or BrickCount.

5.2 newtype

newtype is very similar to defining a new data type with data, however newtype can only have one value constructor. This is used within Figure 4 to define a new type in order to add a new semantic to the language. The newtype definition used here is an example of record syntax, this reduces the amount syntax required in the code, simplifying the solution. Record syntax creates a function which looks up fields of data in a datatype, the function is created in the declaration of the data type. Figure 8 shows how the width function can be defined without using record syntax. A new function has to be defined which takes the newly defined type as input and returns the value stored in the field of the new type. The width function simply returns the value stored in the Width value constructor [6].

Figure 8

```
> newtype Width = Width{width::Int}

==

> newtype Width = Width Int

> width :: Width -> Int
> width (Width widthVal) = widthVal
```

5.3 Instances

Every instance within the DSL implements a specific semantic. This is done by using the newly created type to create an instance of the Block typeclass which states how a the functions to create a block (e.g. brick) are implemented for the specific type/semantic. The definitions of each instances functions almost identically map over to the definitions of algebras used in the deep embedding for the corresponding semantics. This is due to the fact that the shallow embedding also avoids the use of recursion and evaluates functions using the bracketing to determine the order of precedence. The noticeable difference between the deep and shallow embedding here is the types used in the definition. We use the newtype definition to unwrap the type out of it's context in order to transform the value it contains according to the given semantic. We then use the value constructor to wrap the new value in the type to maintain the type of the Block.

5.4 An example

Due to the width function being called, the Width instance of the Block typeclass is entered. The Haskell compiler will only evaluate the statement when the types correctly match across the entire function. Therefor before evaluation begins for the function stated below, the compiler already knows to use the Width instance of the Block typeclass in order for the width function to take in the correct type as input (Width -> Int). Therefor the functions defined in the Width type class are used to evaluate the statement below.

The algorithm for applying a given semantic in the language (Example displayed in Figure 9)

1. Check which instance of the typeclass to use based upon the evaluation function.
2. Apply each function starting with the highest precedence (determined with bracketing), based upon the typeclass instance. The final value is returned after this step.

Figure 9

```
width (beside (brick 1 1) (brick 1 1))

== {- First evaluate the brick functions according to the Width instance. -}
width (beside (Width 1) (brick 1 1))
width (beside (Width 1) (Width 1))
== {- Evaluate the beside function. -}
width (Width (width (Width 1) + width (Width 1))   --width :: Width -> Int,  Width :: Int -> Width
== {- Apply the width functions and evaluate. -}
width (Width(1 + 1))
width (Width(2))
2
```

6 Comparing The Improved Deep And Shallow Embedding

6.1 Similarities

Now there is a improved shallowly and deeply embedded DSL for the Block domain, therefor the strategies for implementing embedded DSLs can be compared once again. First it is important to understand the similarities between the two DSLs. After a quick comparison between the two, it is clear that the implementation of algebras/functions which apply a given semantic to a Block are very similar. In fact the underlying logic is identical between the value constructors for a given function in the deeply embedded DSL and the functions which represent that value constructor within the shallowly embedded DSL. An example is shown below.

```
--Deep and Shallow embedding Width function when a brick is encountered

alg (Brick h w) = 1     --Deep Embedding

brick h w = Width 1     --Shallow Embedding
```

The two snippets of code are taken from the two DSLs implementation of the width semantic for a given Block. If a Block is made of a brick, then the width returned should be 1 as the width of a single brick is 1. This has been implemented almost identically in both DSLs, as for the deep embedding an integer 1 is returned by the alg when Brick is encountered. As for the shallow embedding a Width type for the Block is returned, which has a stored Int of value 1, this means width is equal to 1. It is now clear that the algebras for a given semantic correspond to the typeclass instances within the shallow embedding [1].

Due to this, the two embeddings evaluate in a similar way, starting from the innermost alg application which will be on the base case. The base case of recursion in the Block domain will always be on a brick, as a Block can only be made of bricks. The two embeddings then work up the recursive structure using values from the previous step in order to determine the value of the next step. This continues up until the algebra/functions have been applied to the entire Block structure. The value returned is the correct solution according to semantic applied.

6.2 Differences

Although both types of embeddings now seem to have a lot more in common compared to their previous versions, they still differ in their solutions to implementing the semantics of the Block domain. It is far easier to understand the process the shallow embedding uses to obtain a result. for applying a given semantic to a certain Block. This is shown by the difference in the complexity of proofs in Figures 6 and 9 in the deep and shallow improved embedding sections. The deep embedding requires the usage of cata to apply a given algebra across the recursive Block data structure, in order to find the solution for a given semantic. However the shallow embedding takes advantage of Haskell's type interface, in order to find what instance of functions which create a Block should be used in order to find the solution for a given semantic. In other words, this avoids the need of traversing through the entire data structure of a Block in the deep embedding, in order to swap all In(s) with alg(s).

On the other hand the shallow embedding is more complex on each part of the Block structure it applies a given semantic to. This is shown in Figure 6 and 9, as the deep embeddings alg implementation requires less steps for certain operations. The shallow embedding has to unwrap then manipulate the value stored inside the type then wrap it back in it's original context again, whilst the deep embedding does not need to do this as the value remains in it's context e.g remains Block Int.

7 Conclusion

Now it is clear how to construct and improve embedded DSLs, but why would one go through this process? As discussed earlier DSLs are created to solve a specific problem/domain at hand, rather than using a general purpose languages features to do so. This is because the DSL can express the solution in a clearer way then a general purpose language which is designed to solve problems across many domains. The improvements made to the constructed DSLs in this report have all improved the languages in a way which makes their solution to given semantics more clear. For the deep embedding the recursion was removed from the functions which apply semantics, this allowed for simple algebras to be created which were instead mapped across the entire recursive data structure. These algebras were simple and easy to understand how they arrived at the correct solution. The cata and Fix type provided a generic way to apply these algebras. The shallow embedding was improved so that all the semantic meanings which were required to cover the given domain were available in a single DSL. This made the shallow embedding more practical and generic using typeclasses, allowing more semantic meanings to easily be added if there were any changes to the domain. Essentially the improvements have made the DSLs express solution more clearly, which is exactly their desired purpose.

Another question to ask is, why construct a DSL for two different embeddings? The answer is simple, each has it's own costs and benefits. As discussed earlier for the shallow embedding, it is far easier to understand how the entire language works compared to the deep embedding which requires a large amount of manipulation to a newly created recursive data structure. Therefor, for small domains which are not very complex, a shallow embedding is ideal. However the deep embedding has the advantage that each semantics implemented algebra is simpler and takes less steps to apply, compared to using instances of a typeclass. Also the use of a newly created data structure (abstract syntax tree for the DSL) means optimisations can be made before translating the DSL in to the target language (Haskell). This makes the deep embedding ideal for more complex DSLs, which contain a large number of semantics.

The Block domain used in the figures above is a simple example of a given DSL. This simplicity was chosen to allow the reader to clearly understand the implementation of a given DSL and how it can be improved. The concepts used here can be applied to more complex DSLs written in different host languages. Haskell was chosen to create all of the DSLs in this report over other languages such as C and Java, as it is far easier to understand the solution to the issues presented in creating DSLs. This is similar to why DSLs are used, they express the solution in a clearer way than general purpose languages.

Moving on from this report, there are a variety of topics which can be delved in to. Learning about implementing embedded DSLs in a variety of host languages would further improve the readers understanding of DSLs and the concepts introduced. This can also be done for stand alone DSLs, which require their own compiler. On the other hand a concept known as catamorphism was introduced in the deep embedding, with the use of the cata function. This can be explored in to by further reading on category theory [3], this is a large mathematical topic which has a strong relationship with DSLs.

8 References

[1] Folding Domain-Specific Languages: Deep and Shallow Embeddings, by Jeremy Gibbons and Nicolas Wu, 2014.
[2] Embedded Domain Specific Language, https://wiki.haskell.org/Embedded_domain_specific_language, 2015.
[3] Category Theory, https://en.wikipedia.org/wiki/Catamorphism, 2016
[4] wiki.haskell, Functor Laws, https://wiki.haskell.org/Typeclassopedia#Laws 2015
[5] Bartosz Milewski, Understanding F-Algebras, https://www.schoolofhaskell.com/user/bartosz/understanding-algebras, 2013
[6] Miran Lipovaca, Learn You a Haskell for Great Good, Chapters 3 (Typeclasses 101) & 8 (Typeclasses 102)
[7] Martin Fowler, A description and examples of domain specific languages, http://martinfowler.com/books/dsl.html
[8] data.Fix, An explanation of the Fix data type, https://hackage.haskell.org/package/data-fix-0.0.3/docs/Data-Fix.html

9 Appendices

Semantics covered by the Block domain:

- width: Finds the width of the Block structure, where a Brick always has a width of 1.
- height: Finds the height of the Block structure, where a Brick has a specified height.
- weight: Finds the weight of the Block structure, where a Brick has a specified weight.
- brickCount: Finds the number of Bricks in a Block structure.

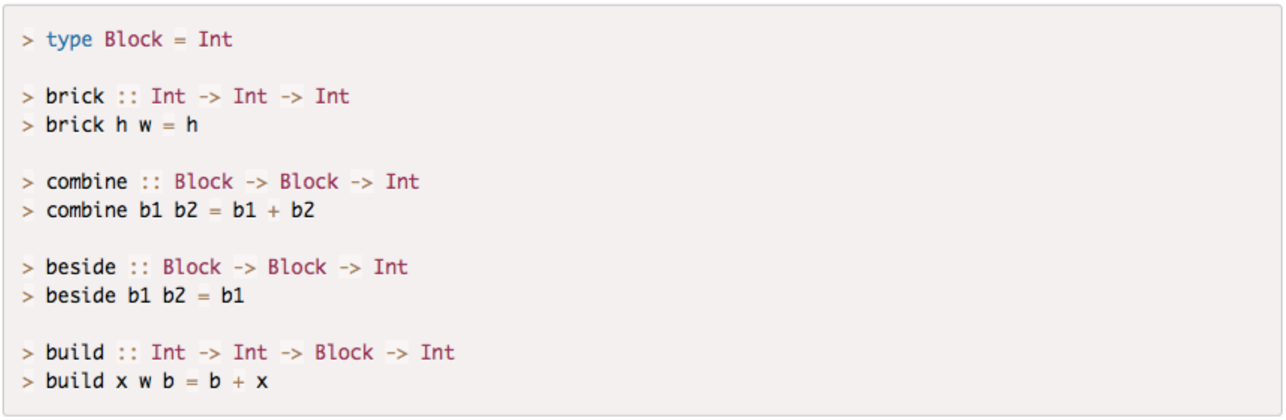
Figure 1 (Extended)

```
 data Block =  Brick Int Int          --Places a single brick of a specified height and weight
>              | Combine Block Block  --Puts one block on top of another (Only if they are the same width)
>              | Beside Block Block   --Puts one block next to another
>              | Build Int Int Block  --Builds up a block by the specified height and of specified weight

--Returns the width of a block

> width :: Block -> Int
> width (Brick x w) = 1
> width (Combine b1 b2) = width(b1)
> width (Beside b1 b2) = width(b1) + width(b2)
> width (Build x w b) = width(b)

--Returns the height of a block

> height :: Block -> Int
> height (Brick x w) = x
> height (Combine b1 b2) = height(b1) + height(b2)
> height (Beside b1 b2) = height(b1)
> height (Build x w b) = height(b) + x

--Returns the total weight of the block

> weight :: Block -> Int
> weight (Brick x w) = x * w
> weight (Combine b1 b2) = weight(b1) + weight(b2)
> weight (Beside b1 b2) = weight(b1) + weight(b2)
> weight (Build x w b) = x * (width b) * w

--Returns the number of bricks in the block

> brickCount :: Block -> Int
> brickCount (Brick x w) = x
> brickCount (Combine b1 b2) = brickCount(b1) + brickCount(b2)
> brickCount (Beside b1 b2) = brickCount(b1) +brickCount(b2)
> brickCount (Build x w b) = brickCount(b) + (width(b) * x)
```

Figure 2.1

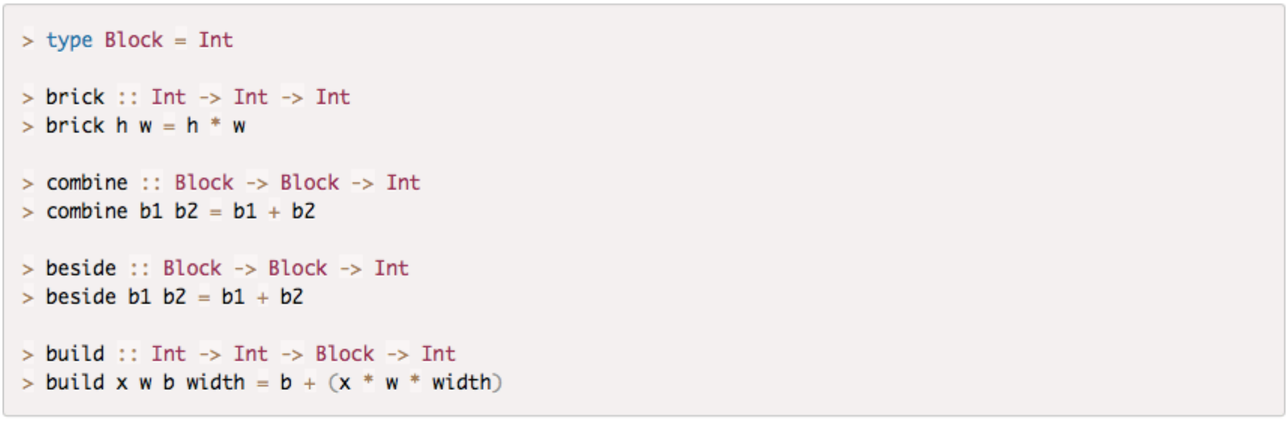
```
> type Block = Int

> brick :: Int -> Int -> Int
> brick h w =  1

> combine :: Block -> Block -> Int
> combine b1 b2 = b1

> beside :: Block -> Block -> Int
> beside b1 b2 = b1 + b2

> build :: Int -> Int -> Block -> Int
> build x w b = b
```

Figure 2.2



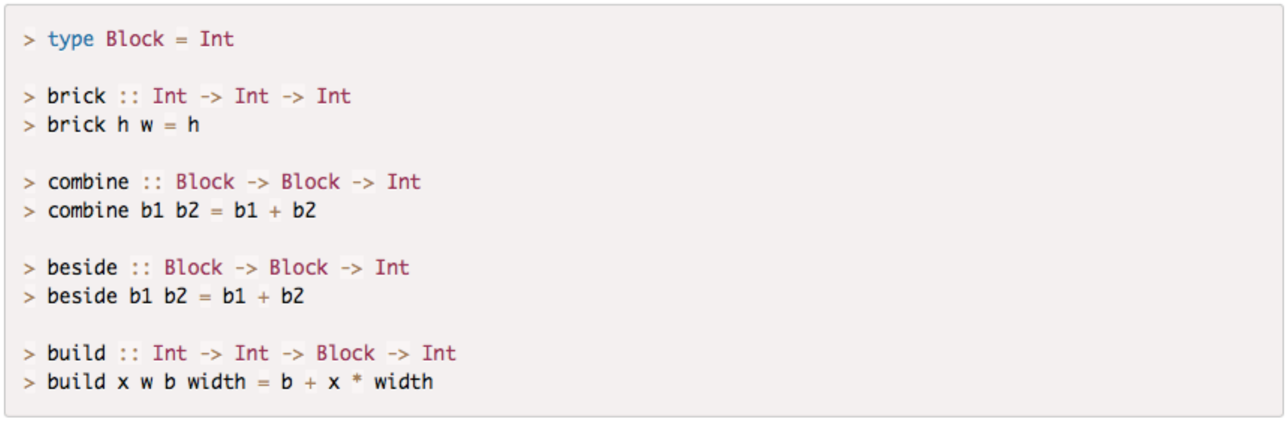
```
> type Block = Int

> brick :: Int -> Int -> Int
> brick h w = h

> combine :: Block -> Block -> Int
> combine b1 b2 = b1 + b2

> beside :: Block -> Block -> Int
> beside b1 b2 = b1

> build :: Int -> Int -> Block -> Int
> build x w b = b + x
```

Figure 2.3



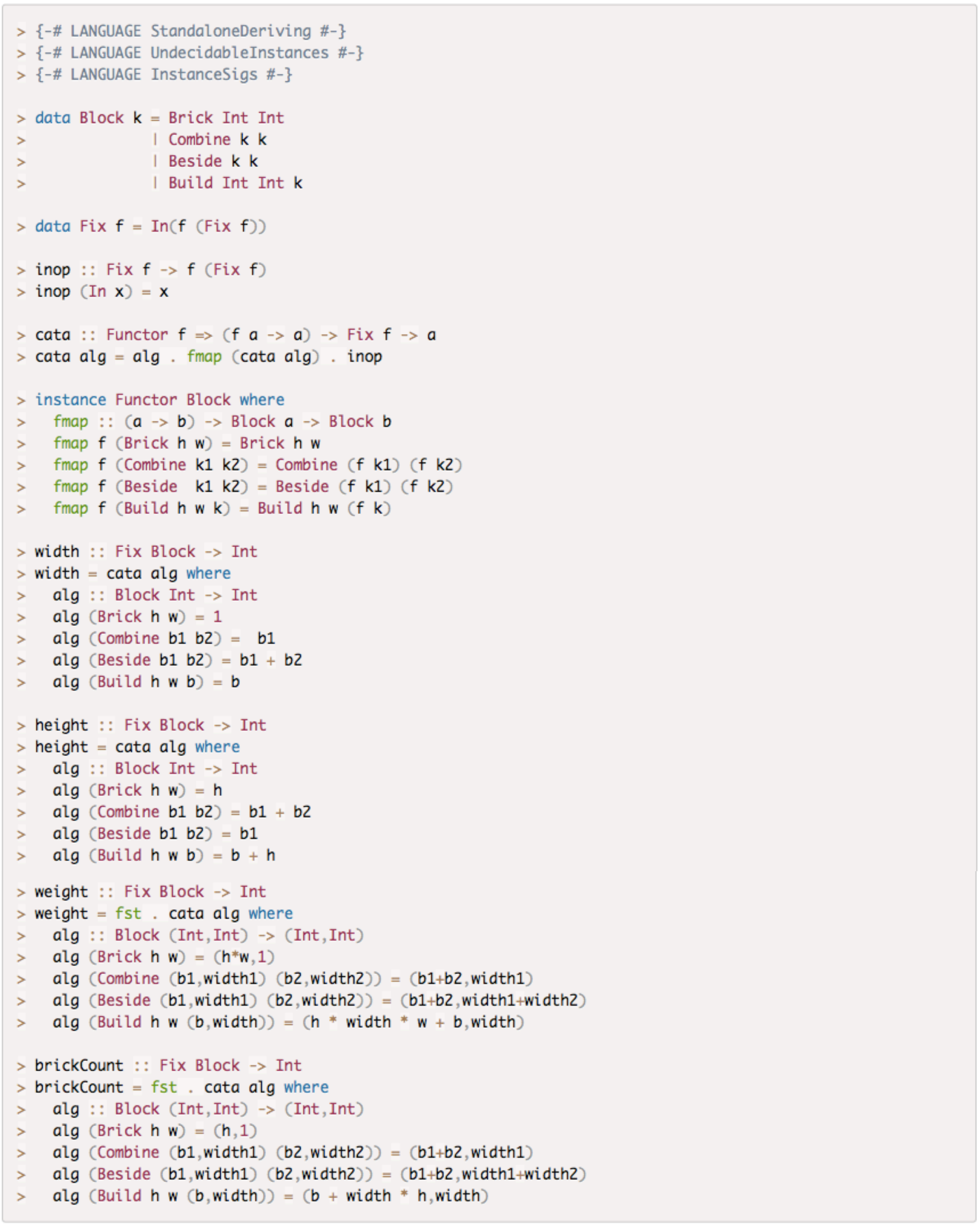
```
> type Block = Int

> brick :: Int -> Int -> Int
> brick h w = h * w

> combine :: Block -> Block -> Int
> combine b1 b2 = b1 + b2

> beside :: Block -> Block -> Int
> beside b1 b2 = b1 + b2

> build :: Int -> Int -> Block -> Int
> build x w b width = b + (x * w * width)
```

Figure 2.4



```
> type Block = Int

> brick :: Int -> Int -> Int
> brick h w = h

> combine :: Block -> Block -> Int
> combine b1 b2 = b1 + b2

> beside :: Block -> Block -> Int
> beside b1 b2 = b1 + b2

> build :: Int -> Int -> Block -> Int
> build x w b width = b + x * width
```

Figure 3 (Extended)



```
> {-# LANGUAGE StandaloneDeriving #-}
> {-# LANGUAGE UndecidableInstances #-}
> {-# LANGUAGE InstanceSigs #-}

> data Block k = Brick Int Int
>              | Combine k k
>              | Beside k k
>              | Build Int Int k

> data Fix f = In(f (Fix f))

> inop :: Fix f -> f (Fix f)
> inop (In x) = x

> cata :: Functor f => (f a -> a) -> Fix f -> a
> cata alg = alg . fmap (cata alg) . inop

> instance Functor Block where
>   fmap :: (a -> b) -> Block a -> Block b
>   fmap f (Brick h w) = Brick h w
>   fmap f (Combine k1 k2) = Combine (f k1) (f k2)
>   fmap f (Beside  k1 k2) = Beside (f k1) (f k2)
>   fmap f (Build h w k) = Build h w (f k)

> width :: Fix Block -> Int
> width = cata alg where
>   alg :: Block Int -> Int
>   alg (Brick h w) = 1
>   alg (Combine b1 b2) =  b1
>   alg (Beside b1 b2) = b1 + b2
>   alg (Build h w b) = b

> height :: Fix Block -> Int
> height = cata alg where
>   alg :: Block Int -> Int
>   alg (Brick h w) = h
>   alg (Combine b1 b2) = b1 + b2
>   alg (Beside b1 b2) = b1
>   alg (Build h w b) = b + h

> weight :: Fix Block -> Int
> weight = fst . cata alg where
>   alg :: Block (Int,Int) -> (Int,Int)
>   alg (Brick h w) = (h*w,1)
>   alg (Combine (b1,width1) (b2,width2)) = (b1+b2,width1)
>   alg (Beside (b1,width1) (b2,width2)) = (b1+b2,width1+width2)
>   alg (Build h w (b,width)) = (h * width * w + b,width)

> brickCount :: Fix Block -> Int
> brickCount = fst . cata alg where
>   alg :: Block (Int,Int) -> (Int,Int)
>   alg (Brick h w) = (h,1)
>   alg (Combine (b1,width1) (b2,width2)) = (b1+b2,width1)
>   alg (Beside (b1,width1) (b2,width2)) = (b1+b2,width1+width2)
>   alg (Build h w (b,width)) = (b + width * h,width)
```

Figure 4

```haskell
fmap id (Combine k1 k2) == Combine k1 k2
{-Proof: First law of Functors-}

fmap id (Combine k1 k2)
=
Combine id(k1) id(k2)
=
Combine k1 k2               --Therefor first law is prooved true.
```

```haskell
fmap (g.f) (Combine k1 k2) == (fmap g . fmap f) (Combine k1 k2)
{-Proof: Second law of Functors-}

= {- Proove by applying fmap on both sides of the equality in the second Law, start with left hand side-}
Combine (g.f)k1 (g.f)k2
=
Combine g(f k1) g(f k2)

= {- Now attempt to equal right hand side to what the left hand side has been simplified to. -}
fmap g (fmap f (Combine k1 k2))
=
fmap g (Combine (f k1) (f k2))
=
Combine g(f k1) g(f k2)     -- right and left sides simpliify to the same value, therefor true.
```

Figure 5

```
      fmap(cata alg)

f (Fix f) -------> f a
    /\              |
    |               | alg
inop|               |
    |              \/
 Fix f    ------->  a
         cata alg
```

Figure 6

```
    width(In(Beside (In(Brick 4 4)) (In(Brick 4 4))))

==  {- width = cata alg -}
    (cata alg) (In(Beside (In(Brick 4 4)) (In(Brick 4 4))))
==  {- Expand cata function -}
    (alg . fmap (cata alg) . inop) ( In(Beside ( In(Brick 4 4) ) ( In(Brick 4 4) )))
==  {- Apply from left to right, apply inop -}
    (alg . fmap (cata alg)) (Beside ( In(Brick 4 4) ) (In(Brick 4 4) ))
==  {- Apply fmap (cata alg), to do this use fmap definition for Beside  (the recursive step) -}
    (alg . (Beside ((cata alg) ( In(Brick 4 4) ))  ((cata alg)  ( In(Brick 4 4) ))))
==  {- Expand left cata function -}
    (alg . (Beside ((alg . fmap(cata alg) . inop) ( In(Brick 4 4) )) ((cata alg) ( In(Brick 4 4) ))))
==  {- Apply inop -}
    (alg . (Beside ((alg . fmap(cata alg)) (Brick 4 4)) ((cata alg) ( In(Brick 4 4) ))))
==  {- Apply fmap(cata alg) using fmap definition for Brick -}
    (alg . (Beside (alg . (Brick 4 4)) ((cata alg) ( In(Brick 4 4) ))))
==  {- Expand cata function -}
    (alg . (Beside (alg . (Brick 4 4)) ((alg . fmap(cata alg) . inop) ( In(Brick 4 4) ))))
==  {- Apply inop -}
    (alg . (Beside (alg . (Brick 4 4)) ((alg . fmap(cata alg)) (Brick 4 4))))
==  {- Apply fmap(cata alg) -}
    (alg . (Beside (alg . (Brick 4 4)) (alg . (Brick 4 4))))
==  {- Now apply the alg function defined for width for each value constructor -}
    (alg . Beside (alg . (Brick 4 4) (1)))
    (alg . (Beside (1) (1)))
    (2)
```

Figure 7 (Extended)

```
> type BlockWidth = Int

> class Block block where
>   brick   :: Int -> Int -> block
>   combine :: block -> block -> block
>   beside  :: block -> block -> block
>   build   :: Int -> Int -> BlockWidth -> block -> block

> newtype Width = Width{width::Int}

> instance Block Width where
>   brick h w           = Width 1
>   combine b1 b2       = b1
>   beside b1 b2        = Width(width b1 + width b2)
>   build h w width b   = b

> newtype Height = Height{height::Int}

> instance Block Height where
>   brick h w           = Height h
>   combine b1 b2       = Height(height b1 + height b2)
>   beside b1 b2        = b1
>   build h w width b   = Height(height b + h)

> newtype Weight = Weight{weight::Int}

> instance Block Weight where
>   brick h w           = Weight w
>   combine b1 b2       = Weight(weight b1 + weight b2)
>   beside b1 b2        = Weight(weight b1 + weight b2)
>   build h w width b   = Weight(weight b + h * width * w)

> newtype BrickCount = BrickCount{brickCount::Int}

> instance Block BrickCount where
>   brick h w           = BrickCount w
>   combine b1 b2       = BrickCount(brickCount b1 + brickCount b2)
>   beside b1 b2        = BrickCount(brickCount b1 + brickCount b2)
>   build h w width b   = BrickCount(brickCount b + h * width)
```

Figure 8

```
> newtype Width = Width{width::Int}

==

> newtype Width = Width Int

> width :: Width -> Int
> width (Width widthVal) = widthVal
```

Figure 9

```
width (beside (brick 1 1) (brick 1 1))

== {- First evaluate the brick functions according to the Width instance. -}
width (beside (Width 1) (brick 1 1))
width (beside (Width 1) (Width 1))
== {- Evaluate the beside function. -}
width (Width (width (Width 1) + width (Width 1))   --width :: Width -> Int,  Width :: Int -> Width
== {- Apply the width functions and evaluate. -}
width (Width(1 + 1))
width (Width(2))
2
```