Formation of Embedded DSLs in Haskell

Stefan Klas

1 Introduction

In the past the creation of new programming languages for a particular domain of use was a time consuming task requiring a specialist in language engineering. Over time, the need for more specialized languages has grown. As better tools and techniques have become available including functional programming, the creation of domain specific languages(DSLs) has become quicker and easier. We therefore explore the development of a new domain specific language from discussing the DSL concept, defining data structures and grammar to implementing a parser and DSL evaluation code in the functional programming language Haskell.

2 Domain Specific Language

DSLs are languages that are designed specifically to express solutions to problems in a particular domain. Unlike General Purpose Languages(GPLs) like Java, functionality is limited and there is no guarantee of Turing Completeness. They consist of expressive notions and constructs that make it intuitive to program for specialists working in that domain. Popular DSLs include CSS for style sheets and SQL to deal with relational data but many more exist in various domains like Jnario for writing readable acceptance specifications and MLFi for describing and pricing financial contracts.

The key benefit DSLs offer over GPLs is the high level of abstraction to the user. Reducing the clutter of low level implementation details allows users to focus on solving the problem at hand aligned with the particular domain which significantly increases productivity. Furthermore users don't need to be expert programmers because the syntax is far smaller and uses expressive representation that allows for code generation.

However DSLs come with their problems. First, to really be able to create your own DSL, the developer must have a deep understanding of the domain in which it is intended to be used. Secondly they have a high maintenance effort, due to the relatively low usage and so some DSLs like LN for geometry problems are not supported for long. Thirdly there is a trade off between more approachable and more powerful [2].

A DSL can either be standalone or embedded. A standalone DSL means you create from scratch your own syntax and semantics of the language as well as the technique to translate this into object code, through either building a compiler or an interpreter. The alternative is to embed the DSL within a host GPL and make use of the existing facilities and infrastructure of the GPL such as editors, compilers and debuggers [1]. In this paper we will focus solely on an embedded implementation in Haskell. This brings with it significant advantages and also a few disadvantages. The advantages include a reduced development process and increased features through piggy backing on the Haskell infrastructure. The disadvantages involve having to write in the syntax of the host language which can be daunting for non-functional experts and the lack of IDE support for the new DSL being designed (meaning the editor has no awareness of its grammar and constraints apart from what the type system can offer).

3 Sandwich Making DSL

This paper will show the process of constructing a DSL inside the sandwich making domain. The language will allow the construction of valid sandwiches with different fillings.

3.1 Data Structure

There exist two methods to embed a DSL within a GPL which are shallow embedding and deep embedding [11]. Shallow embedding maps terms in the DSL directly to their semantics while deep embedding will construct an abstract syntax tree, which can then be traversed for evaluation. The deep embedding approach is favoured as it allows for easy formulation of multiple interpretations of the data structure, without having to change large chunks of code and type signatures throughout. These interpretations could be the likes of calculating calorie

count of the sandwich or determining whether it is vegetarian or finding the cost of making the sandwich all based on the abstract syntax tree.

Lets first define our data structure and make the Filler data type part of the Functor typeclass.

```
data Sandwich = Sandwich Bread Inside Bread
data Inside = Inside Spread (Fix Filler) Sauce
data Spread = Butter
data Filler k = Add k k | Ham | Cheese | Lettuce | Chicken
data Sauce = Ketchup | Mayonnaise
data Bread = Bread

instance Functor Filler where
    fmap f (Add x y) = Add (f x) (f y)
    fmap f Ham = Ham
    fmap f Cheese = Cheese
    fmap f Lettuce = Lettuce
    fmap f Chicken = Chicken
```

As can be seen, a sandwich will consist of two slices of bread which are separated by an inside. The inside must start with butter, then contain one or more fillings followed by a sauce. The data type definitions should be relatively straightforward except for possibly the way the recursive fillings are represented through the use of a Fix data type which is a fixed point combinator. To get a deeper understand of this, lets look at Fix's defintion.

```
data Fix f = In (f (Fix f))
```

One can initially spot that the data type contains some recursion as `Fix f` can equal `In (f (Fix f))` as well as `In (f (In (f (Fix f))))` and so on.

By replacing f with `Filler`, the fillings the sandwich can have are described as `Fix Filler = In (Filler (Fix Filler))`. By matching the types with the Filler type constructor, we can conclude that k is of type `Fix Filler` in this case. Therefore using the definition, `Fix Filler` could result in `In (Add (Fix Filler) (Fix Filler))` or `In (Ham)`. But now the two arguments in the Add data constructor can further recurse and so form a tree containing multiple fillings where the leaf nodes are the nullary data constructors. A valid `Fix Filler` would be `In (Add (In Ham) (In Cheese))`.

Is it worth the trouble of defining the fillings in this way? The alternative would have been a simple recursive definition like `data Filler = Add Filler Filler`. By abstracting away the recursion, a catamorphism can be used to fold through the fillings tree. The beauty of this is that unlike the alternative, it requires very little thought where the algorithm needed considers only one level in the tree and the catamorphism takes care of the rest.

```
cata :: (Functor f) => (f b -> b) -> Fix f -> b
cata alg (In x) = alg (fmap (cata alg) x)
```

In essence, the catamorphism works by recursively applying itself to the children in the tree. Once it has reached the leaf nodes, it can start evaluating using the algorithm given and work its way back up to the top to give a final result. Converging occurs because it applies fmap to subtrees that are strictly smaller than the original tree.

To give an example and show the steps involved in traversing through the valid Fix Filler `In (Add (In Ham) (In Cheese))`, first lets define an example algorithm for some of the Filler data constructors.

```
alg Ham  = 30
alg Cheese = 20
alg Add x y  = x + y
```

Then using a catamorphism on the Fix Filler expression `In (Add (In Ham) (In Cheese))`, the following occurs:

```
cata alg In (Add (In Ham) (In Cheese))
        = alg (fmap (cata alg) Add (In Ham) (In Cheese))
        = alg (Add (cata alg (In Ham)) (cata alg (In Cheese)))
        = alg (Add (alg (fmap (cata alg) Ham))(alg (fmap (cata alg) Cheese)))
        = alg (Add (alg Ham)(alg Cheese))
```

Here it has now reached the leaf nodes

```
        = alg (Add 10 20)
        = 30
```

All of the leaf nodes in the tree have been added up. Expanding the algorithm to include all Filling data constructors will be done in the Evaluation Stage 3.5.

3.2 Parser

Having defined the data structure, the next step in DSL development is to make the parser using parser combinators that will parse a valid input into an abstract syntax tree that is represented by the data structure.

In order to parse input, two usual phases are required. The first is the lexical analysis, which breaks the input into a series of tokens, removing any white spaces. The second step is the syntax analysis which compares the tokenised input with the production rules in the grammar to detect any errors and otherwise produce the tree.

Using parser combinators is a quick and easy way of building up functional parsers and allows these phases to be combined. A parser combinator is a higher-order function (taking in one or more functions to return another function) that accepts several parsers as input and returns a new parser as its output. This means you can start with very basic parsers such as the fail parser or character parser and build up far more complicated ones.

First, a grammar for the language is needed to define the syntax that is allowed. An input like bread + butter + ham + cheese + ketchup + bread will be valid whilst bread + ham + bread will not. A sandwich must start with bread then butter then one or more fillings followed by sauce and topped off with bread.

BNF Grammar:

⟨*sandwich*⟩ ::= '`bread`' ⟨*addop*⟩ ⟨*inside*⟩ ⟨*addop*⟩ '`bread`'
⟨*inside*⟩ ::= '`butter`' ⟨*addop*⟩ ⟨*filler*⟩ ⟨*addop*⟩ ⟨*sauce*⟩
⟨*filler*⟩ ::= ⟨*ingred*⟩ [⟨*addop*⟩ ⟨*ingred*⟩]*
⟨*ingred*⟩ ::= '`ham`' | '`cheese`' | '`lettuce`' | '`chicken`'
⟨*sauce*⟩ ::= '`mayonnaise`' | '`ketchup`'
⟨*addop*⟩ ::= '`+`'

There are many different approaches to writing parsers in Haskell such as writing all the parser combinators from scratch or making use of combinator libraries like Parsec [4]. To gain a deeper understanding of how basic parsers are constructed and give a clearer overall picture, this paper will opt for the former over the latter.

To begin the parser construction, a definition of the parser is required. The parser is in essence a function that takes a string and returns a list of tuples which is wrapped in a Parser constructor. On parsing, each tuple formed would contain the parsed part and the remaining string that is left over. Using the parse function will allow for the function inside the Parser to be exposed so that input can be fed in.

```
data Parser a = Parser  (String ->[(a, String)])
parse (Parser a) = a
```

The convention is that if a singleton list is formed at the end of parsing, it has been successful and if the list is empty it has failed. Returning a list also opens up the possibility of having a number of different tuples which could occur if the grammar is ambiguous [7].

The parser needs to be made instance of Functor, Applicative and Monad typeclasses.

```
instance Functor Parser where
  fmap f (Parser cs) = Parser (\s -> [(f a, b) | (a, b) <- cs s])

instance Applicative Parser where
  pure = return
  (Parser c) <*> (Parser d) = Parser (\s -> [(f a, s2) | (f , s1) <- c s, (a, s2) <- d s1])

instance Monad Parser where
  return a = Parser (\cs -> [(a, cs)])
  p >>= f =  Parser (\cs -> concat [(parse (f a)) cs' | (a,cs') <- (parse p) cs])
```

Regarding making the Parser a Monad, later extensive use will be made of the do notation to bind parsers together, forming more complicated ones. The bind operation p >>= f works by forming a new parser that will start by parsing the string using p parser. For each tuple formed, the resulting parsed part is fed into f to form another parser that then parses on the leftover string. The Parser is made an instance of Applicative and Functor typeclasses such that it can be an instance of the Alternative typeclass.

The parser is then also made an instance of MonadPlus and Alternative typeclasses.

```
instance MonadPlus Parser where
  mzero = Parser (\cs -> [])
  mplus p q = Parser (\cs -> (parse p) cs  ++ (parse q) cs)

instance Alternative Parser where
  empty = mzero
  (<|>) = option

option :: Parser a -> Parser a -> Parser a
option p q = Parser (\cs -> case parse p cs of
                                [] -> parse q cs
                                res -> res)
```

Being part of the MonadPlus class adds two more operations, namely mzero and mplus and means that Parser must adhere to the MonadPlus rules. mzero is the fail Parser, where when given a string it will always return an empty list. An essential rule from the class is that mzero propagates through a bind: mzero >>= f = mzero and f >>= mzero = mzero. To making use of this, the Parser will return mzero when parsing fails resulting in the failure being unrecoverable. mplus also has its use in combining two parsers into one.

Being part of the Alternative class also adds a choice operator. p <|> q will mean trying to parse the string using the p parser first. If the result fails (empty list), it will then try the second parser, otherwise return. As the result is a Parser itself, it is possible to chain these up to given greater choice between more parsers.

The next step is defining some basic parser combinators following ideas described in [5] and [6].

One of the most basic but essential parsers needed is the character parser. Given a string, this will parse the first character if it exists.

```
item :: Parser Char
item = Parser $ \s ->
  case s of
   []     -> []
   (c:cs) -> [(c,cs)]
```

In order to apply a particular parser multiple times, the many method can be used to form a new parser that parses a list of these things. For example, feeding item into many will give a Parser that parses many characters. The actual definition for many and many1 can look a bit confusing because they use each other in their definitions. In essence how it works is that for many1, the given parser will be applied to the string, storing the parsed part into a. It will then repeat the same process on the leftover string, until it's unable to go further and a Parser [] is returned. It then builds up the parsed parts by unravelling the recursion.

```
(+++) :: Parser a -> Parser a -> Parser a
p +++ q = Parser (\cs -> case parse (p `mplus` q) cs of
                                [] -> []
                                (x:xs) -> [x])

many :: Parser a -> Parser [a]
many p = many1 p +++ return[]

many1:: Parser a -> Parser [a]
many1 p = do{ a <- p; as <- many p; return (a:as);}
```

Satisfy builds on the item Parser defined above, parsing a character and checking it against the character to boolean function given. If it returns false, mzero is returned which ensures that a parse error occurs.

```
satisfy :: (Char -> Bool) -> Parser Char
satisfy p = do{ c <- item; if p c then return c else mzero}
```

Building on satisfy, oneOf will parse a character if it is one of the elements in the list of characters given. Failure will again result in mzero being returned and parse error occuring.

```
oneOf :: [Char] -> Parser Char
oneOf s = satisfy (flip elem s)
```

Chain1 will parse using the first argument p at least once but otherwise repetitively whilst separated by the second argument which parses an operator that is used to combine the parser p results. It will continuously do this until no more operators exist.

```
chainl1 :: Parser a -> Parser (a -> a -> a) -> Parser a
p `chainl1` op = do {a <- p; rest a}
  where rest a = (do f <- op
                     b <- p
                     rest (f a b))
                 <|> return a
```

3.3 Complex Combinators

The Parsers defined so far are very basic but are important building blocks to form more complicated recursive and lexical combinators that we need.

```
--Parser that parses a certain character
char :: Char -> Parser Char
char c = satisfy (c==)

--Function that given a string creates a parser that parses that string
string :: String -> Parser String
string [] = return []
string (c:cs) = do { char c; string cs; return (c:cs)}

--Parser that will parse as many new lines, empty spaces or tabs as it can
spaces :: Parser String
spaces = many (oneOf " \n\r")

--Function that given a parser p, forms a new parser that applies p first and then parses all
↪  trailing spaces that follow
token :: Parser a -> Parser a
token p = do { a <- p; spaces ; return a}

--Function that given a string forms a new parser that parses that string and all trailing
↪  spaces
reserved :: String -> Parser String
reserved s = token (string s)
```

3.4 Sandwich Parser

Now all tools exist to create our specific sandwich parser. It is important to emphasize that up to now, the parsers defined are not domain specific but polymorphic and so can be used when creating many different parsers in Haskell. Defining the sandwich parser now becomes very easy as the structure very much reflects the grammar we outlined above.

Sandwich is a parser, that contains several Parsers combined by bind. It will first try to parse a given string using the bread parser. If it is successful, it will use the addop parser on the leftover string, then inside parser, addp parser and finally bread parser again. If it succeeds in all of this, the list returned will be a singleton tuple containing a Sandwich data type and the leftover string.

```
--Parser to parse the whole sandwich
sandwich :: Parser Inside -> Parser Sandwich
sandwich m = do { bread;
                  addop;
                  n <- m;
                  addop;
                  bread;
                  return (Sandwich Bread n Bread);
                }
--Parser to parse the inside, consisting of a spread, a number of fillings and a sauce
inside :: Parser Inside
inside = do { s <- spread;
              addop;
              t <- filler;
              addop;
              u <- sauce;
```

```
            return (Inside s t u)
          }

--Helper function that creates a parser for all terminal symbols in the grammar except the
↪  plus symbol
assoc :: String -> a -> Parser a
assoc x f = reserved x >> return f

--Parser to parse the bread
bread :: Parser Bread
bread =  assoc "bread" Bread

--Parser to parse the butter
spread :: Parser Spread
spread = assoc "butter" Butter
```

Sauce makes use of the choice operator, trying to parse the word ketchup first and on failure, trying with mayonnaise.

```
--Parser to parse the sauce
sauce :: Parser Sauce
sauce = (assoc "ketchup" Ketchup) <|> (assoc "mayonnaise" Mayonnaise)

--Parser to parse one or more fillings
filler :: Parser (Fix Filler)
filler = ingred `chainl1` addop

--Parser to parse one ingredient
ingred :: Parser (Fix Filler)
ingred = (assoc "ham" (In Ham)) <|> (assoc "cheese" (In Cheese)) <|> (assoc "lettuce" (In
↪  Lettuce)) <|> (assoc "chicken" (In Chicken))

--Parser to parse the plus symbol
addop :: Parser ((Fix Filler) -> (Fix Filler) -> (Fix Filler))
addop  = (infixOp "+" (((.) . (.)) In Add))

--Helper function that creates the parser for plus symbol
infixOp :: String -> (a -> a -> a) -> Parser (a -> a -> a)
infixOp x f = reserved x >> return f
```

Having defined all the parsers, it is possible to now run the sandwich parser on a given string. In doing so, 3 different situations could arise that need to be taken account of:

1. If the tuple inside the singleton list returned has no string left over, we know that parsing has been successful and therefore we can return the sandwich data structure.

2. If the tuple still contains some leftover string that hasn't been parsed, it means that parsing has been successful up to that point but could not continue and so an error should occur.

3. If the list returned is empty, then parsing failed and mzero must have propagated through the binds so again an error should occur.

These 3 cases are dealt with below

```
runParser :: Parser Sandwich -> String -> Sandwich
runParser m s = case parse m s of
                [(res, [])] -> res
                [(_, rs)] -> error "Did not consume all of string"
                _           -> error "Parse error"
```

Having formed a working parser, it can now recognise a valid sandwich input and return an abstract syntax tree. The final step is traversing through this tree and applying meaning.

3.5 Evaluation

There are many different "observations" that can be applied to the sandwich constructed. An observation refers to a function over the sandwich datatype. A scenario is a person working in a sandwich shop wanting to calculate the number of calories in a sandwich. For this case, a function is needed which cycles through each ingredient and accumulates the total calorie count. The function works by pattern matching on each data type and using the catamorphism discussed earlier to fold the fillings tree.

```
calorie (Sandwich b1 i b2) = (calorie_br b1) + (calorie_i i) + (calorie_br b2)
        where
           calorie_br (Bread) = 30
           calorie_i (Inside x y z) = (calorie_bu x) + (calorie_f y) +(calorie_s z)
                      where
                        calorie_f :: (Fix Filler) -> Int
                        calorie_f y = cata calorie_f1 y where
                          calorie_f1 Ham = 30
                          calorie_f1 Lettuce = 10
                          calorie_f1 Cheese = 20
                          calorie_f1 Chicken = 40
                          calorie_f1 (Add x y) = x + y
                    calorie_s Ketchup = 20
                    calorie_s Mayonnaise = 10
                    calorie_bu (Butter) = 20
```

All that needs to be done now is defining the function main and the sandwich making DSL is fully functional!

```
run s = runParser ((sandwich inside)) s

main :: IO()
main = forever(
        do {
          putStr ">>>>";
          a <- getLine;
          print ("Calories: " ++ show(calorie (run a)));
        })
```

Here are some examples of it working:

bread + butter + lettuce

>> Parser error

bread + butter + lettuce + cheese + chicken + lettuce + ketchup + bread

>> Calories: 180

4 DSL using Free Monad and Interpreter Pattern

The Free Monad Interpreter Pattern is a lightweight way of generating a DSL program as pure data with do-notation and then running it on an interpreter, in essence modelling imperative programming [7]. Unlike the constructed monadic functions above that hide the underlying interpreter, here we are separating the specification from the implementation. This means it's much easier to alter the interpretation if we wanted.

The parser above also lacked accurate error messages telling the user exactly why the parsing failed. Ideally the user who tries to put chicken after ketchup should get an error saying this isn't allowed. However implementing this would have been a little tricky as once failure occurs (mzero), due to the binds that follow, one couldn't assess the situation and throw an error. Luckily because of the recursive nature of the interpreter here, this is much easier, taking advantage of monad transformers. Monad transformers are similar to regular monads but are not standalone entities. Instead they allow for increased functionality of a monad such as dealing with error handling, state handling, write handling etc [9].

Also used here is the Free Monad which is a different way of producing a syntax tree configured by some "signature functor" which is f.

```
data Free f a  = Pure a | Free (f (Free f a))
```

Very much like Fix there are two data constructors. The recursive Free data constructor allows for the formation of branches in the tree and Pure forms the leaf nodes.

An important element in understanding how Free Monad works is looking at its monad definition.

```
instance Functor f => Monad (Free f) where
        return x = Pure x
        (Pure r) >>= f = f r
        (Free x) >>= f = Free (fmap (>>= f) x)
```

The return definition should be intuitive but the hidden feature about bind operations is that it allows for the growing of trees by replacing leaf nodes. Lets explore this in more detail with an example.

Lets define an expression data type which can either be a value or a plus operation and make it an instance of Functor typeclass.

```
data Expr a = Val a | Plus a a
instance Functor Expr where
         fmap f (Val a) = Val a
         fmap f (Plus x y) = Plus (f x) (f y)
```

Then using the first bind definition with Pure on the left side yields the following.

```
growleaf :: Free Expr Int
growleaf = Pure 3 >>= (\a -> Free (Plus (Pure a) (Pure a)))
         = Free (Plus (Pure 3) (Pure 3))
```

What this shows is that the value of the leaf node has been replaced by the tree containing the value. Taking the second bind definition with Free on the left side results in the following piece of code.

```
growtree :: Free Expr Int
growtree = Free (Plus (Pure 3) (Pure 2)) >>= f
         = Free (fmap (>>=f) (Plus (Pure 3) (Pure 2)))
         = Free (Plus (Pure 3 >>= f) (Pure 2 >>= f))
```

This shows that both left and right leaf nodes are being replaced by f so there is growth on both branches.

How can all this be used? We are going to create a new data structure called Ingredient to represent our sandwich. We have combined some of the previous components like Bread and Butter into one to make our code shorter for this paper. Each data constructor contains next which will contain all the following instructions that need to be carried out. The way the rest of the instructions get into next is using the bind, exactly like described above.

```
data Ingredient next = Bread_Butter Int next    |  Ham Int next
                     |  Cheese Int next      |  Lettuce Int next
                     |  Chicken Int next     |  Ketchup_Bread Int next
                     |  Mayonnaise_Bread Int next

instance Functor Ingredient where
  fmap f (Bread_Butter num next) = Bread_Butter num (f next)
  fmap f (Ham num next) = Ham num (f next)
  fmap f (Cheese num next) = Cheese num (f next)
  fmap f (Lettuce num next) = Lettuce num (f next)
  fmap f (Chicken num next) = Chicken num (f next)
  fmap f (Mayonnaise_Bread num next) = Mayonnaise_Bread num (f next)
  fmap f (Ketchup_Bread num next) = Ketchup_Bread num (f next)

--liftF command = Free (fmap Pure command)
bread_butter = liftF (Bread_Butter 50 ())
ham = liftF (Ham 30 ())
cheese = liftF (Cheese 20 ())
lettuce = liftF (Lettuce 10 ())
chicken = liftF (Chicken 40 ())
ketchup_bread = liftF (Ketchup_Bread 50 ())
mayonnaise_bread = liftF (Mayonnaise_Bread 40 ())

prog = do
        bread_butter
```

```
        chicken
        lettuce
        cheese
        lettuce
        ketchup_bread
```

So how does it all work? The do-notation in prog first has the instruction bread_butter which looks like `Free (Bread_Butter 50 (Pure ()))`. When we bind this with the second instruction chicken which is `Free (Chicken 40 (Pure ()))` we get the following:

```
Free (Bread_Butter 50 (Pure ())) >>= chicken
    = Free (fmap (>>= chicken) Bread_Butter 50 (Pure ())
    = Free (Bread_Butter 50 (Pure () >>= chicken)
    = Free (Bread_Butter 50 (Pure () >>= \_ -> Free (Chicken 40 (Pure ()))))
    = Free (Bread_Butter 50 (Free (Chicken 40 (Pure ()))))
```

So as can be seen, the next part of the data type which was Pure () was replaced by another Free data type. This carries on for all the instructions that follow to form a long structure of type Free Ingredient ().

The idea of the interpreter is now to unravel all the instructions that are contained in the Free Monad. To make the interpreter more sophisticated, a State Monad [10], with a map state, is used to store all the instructions encountered so far. The map maps possible ingredients to a list of integers, the first being the amount of this ingredient seen and the second the current calorie total for this ingredient. This would allow for same ingredients to have different calorie counts. The monad transformer ExceptT is wrapped around the State Monad to allow for specific errors to be thrown, based on the current state. For example, if a filling is added without any bread and butter being in the current state, an exception is thrown saying "Missing bread and butter".

```
type MapStringInts = Map String [Int]
```

We define all the possible errors that could occur.

```
data ParserError = BreadButterError | FillerError | SauceBreadError | Incomplete
instance Show ParserError where
  show BreadButterError = "Only 1 bread and butter allowed"
  show FillerError = "Missing bread and butter"
  show SauceBreadError = "Missing fillings and/or bread and butter"
  show Incomplete = "Missing sauce and bread"
```

The interpreter, given the instructions wrapped in the Free monad, will call its helper function interpret' to recursively go through the instructions and update the state. Running runExceptT will then unwrap the ExceptT monad transformer with the value in the state now being of type Either. If Left value is returned, failure has occured and an error message is printed, otherwise the sandwich produced is printed to console.

```
interpret :: Free Ingredient a -> IO ()
interpret prog =
    case runState (runExceptT (interpret' prog)) empty of
        (Left error, _) -> putStr (show error ++ "\n")
        (Right _ , state) -> putStr (a ++ "Calorie count :" ++ show b ++ "\n")
            where
            (a,b) = foldWithKey (\k a b -> (k ++ ": x" ++ show (head a) ++ "\n" ++ fst b, snd
↪  b + a !! 1))  ("", 0) state
```

A sandwich must only have one bread and butter, which comes right at the beginning. As a result, when the interpret' pattern matches with Bread_Butter instructions, it checks whether it has already occurred. If it has, it throws an exception and otherwise updates the state before moving on to the next instruction.

```
interpret' :: Free Ingredient a -> ExceptT ParserError (State MapStringInts) ()
interpret' (Free (Bread_Butter num next)) =
    do state <- get
       case (Map.lookup "bread_butter" state) of
         Just (value) -> throwError BreadButterError
         Nothing -> do
                     modify (\s -> insert "bread_butter" ([1,num]) s)
                     interpret' next
```

Pattern matching on Ketchup_Bread, interpret' checks that both Bread_Butter and at least one filling have already been added to the state. Without this, it isn't a valid sandwich and so a SauceBreadError is thrown.

Otherwise it returns, ending the recursion which means that any instruction that follows is ignored as a valid sandwich has been produced. A very similar definition is needed for Mayonnaise_Bread.

```
interpret' (Free (Ketchup_Bread num next)) =
    do state <- get
            if ((foldWithKey (\k a b -> not(null a)) False (filterWithKey (\k _ -> k /=
↪   "bread_butter") state)) && isJust(Map.lookup "bread_butter" state))
            then    do
                      modify (\s -> insert "sauce_bread" ([1,num]) s)
                      return ()
            else throwError SauceBreadError
```

Pattern matching on Ham, interpret' needs to check with the ham filling that bread and butter has been added beforehand. Multiple slices of ham being added updates the state to increase the ham count and the total ham calories. Again very similar definitions are needed for the other fillings.

```
interpret' (Free (Ham num next)) =
    do  state <- get
        case (Map.lookup "bread_butter" state) of
                      Nothing -> throwError FillerError
            Just (value) -> case (Map.lookup "ham" state) of
                    Just (x:y:xs) -> do
                                        modify (\s -> insert "ham" ([x+1,y+num]) s)
                                        interpret' next
                    Nothing -> do
                                  modify (\s -> insert "ham" [1,num] s)
                                  interpret' next
```

Lastly, if the next instruction doesn't pattern match with the above it is Pure (). We know that this isn't a valid sandwich as it is definitely missing a sauce and bread which would not let this occur so Incomplete error is thrown.

```
interpret' _ = throwError Incomplete
```

Interpreting the above function prog in the console gives the following output:

>>bread_butter: x1

>>cheese: x1

>>chicken: x1

>>lettuce: x2

>>sauce_bread: x1

>>Calorie count: 180

If we forget to add a ketchup_bread or mayonnaise_bread in the prog, the console output is this:

>>Missing sauce and bread

In case we omit to state any fillings, the console output is the following message:

>>Missing fillings and/or bread and butter

5 Conclusion

To conclude, in Section 2 to 3, we have discussed the full implementation requirements to make a new embedded DSL in Haskell (for the domain of sandwich making). After having defined a DSL through a BNF grammar, we have created parser combinators to achieve a parser that can parse input from the console and creates an abstract syntax tree based on the language definition. In addition, we have implemented an evaluation which works based on this syntax tree.

In section 4, we applied a second approach. We used a more general technique of implementing a very similar DSL by emulating an imperative programming style in Haskell using Free Monads. Instead of fetching the input from the console, the input is provided programmatically via a do-notation. The validity of the sandwich definition is now checked step by step using a State Monad which is essentially associated with the now slightly modified language grammar.

References

[1] Nicolas Wu, Jeremy Gibbons. *Folding Domain-Specific Languages: Deep and Shallow Embeddings*

[2] Debasish Ghosh. *DSLs in Action (Chap 1-4)*

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

[4] Bryan O'Sullivan, Don Stewart, and John Goerzen *Real World Haskell (Chap 18)*

[5] Stephen Diehl *Parsing - http://dev.stephendiehl.com/fun/002_parsers.html*

[6] Graham Hutton, Erik Meijer *Monadic Parsing in Haskell*

[7] Gabriel Gonzalez *Why free monads matter - http://www.Haskellforall.com/2012/06/you-could-have-invented-free-monads.html*

[8] Tikhon Jelvis *What is the Free Monad + Interpreter Pattern - http://softwareengineering.stackexchange.com-/questions/242795/what-is-the-free-monad-interpreter-pattern*

[9] Martin Grabmull *Monad Transformers Step by Step Oct 2006*

[10] Brandon Si *The State Monad: A Tutorial for the Confused - http://brandon.si/code/the-state-monad-a-tutorial-for-the-confused/*

[11] Jeremy Gibbons. *Functional Programming for Domain-Specific Languages*