Final Tagless While Language, An Exercise in Mentioning the Unmentionable

Jeff Young

1 System Overview

This project implement's the WHILE Language using the final tagless style in just two haskell files, CoreLang.hs and Extensions.hs. Final tagless style, while briefly discussed in class, is essentially a semantic-based implementation of whatever is being made. In Haskell, this translates into a typeclass based approach for the WHILE language. For example, instead of having a Abstract data type for statements in the language, we will instead have a typeclass with functions that represent statements in our language. In this sense, final tagless allows extensibility in the constructor dimension, because one may extend value constructors and make a new instantiation of the type class, and it also allows extensibility in the operator dimension, because one is free to write their own typeclass in final tagless style, instantiate and mix it with the core system. An important note is that final tagless style is also extensible in a third dimension, the evaluator dimension, because all of the language constructs are functions in a typeclass, one is free to implement them how they please. Hence, one may choose to implement them with strict evaluation, or lazy evaluation, with a global state or not.

1.1 The Core System

The core system typeclass definitions are to be found in the CoreLang.hs file. The file holds all typeclass definitions and instantiations for the core language. These typeclasses support a minimal implementation of the WHILE language. Specifically, these are boolean expressions, arithmetic expressions, and statements; all of their types are self-explanatory. The while language, in it's core form, only supports integers, booleans, and a unit as primitives.

1.2 The Extensions to the Core

The extensions made to the core language are in the Extensions.hs file and are as follows. In the constructor domain I extend the language with Strings, Floats, and a list constructor. In the operator dimension I extend the language with two compatible extensions, a trivial extension, adding exponentials to the language, and adding a printing operator to the language. The harder, non-compatible extensions, include operators for strings, floats, and lists. In this way the core language is extended in both the constructor dimension, and operator dimensions. It should be noted that for every implementation I instantiate the core language with a state monad thereby "extending" the while language with a global state (which is normally assumed). This suggests that the final tagless approach is flexible enough to incorporate any monad or monad transformer stack. While I did not explore the "third dimension" of extensibility this project proves that it is possible.

2 Extensibility Scenario

The Extensibility Scenario for the WHILE language is identical to that of any embedded DSL. The scenario is generally this: I, the EDSL author want to write my EDSL in a static, strongly typed language like haskell. Then I want to be able to share that EDSL with anyone on the internet and have them use it. This requires that I've taken time to consider how my users may use my EDSL. But it is impossible to predict every use case, so by writing my EDSL in a final tagless style I can ensure that people who do not have access to the source code (because I'll restrict access) can still extend and use the EDSL however they deem fit. In this project's specific case the core WHILE language is the EDSL, and some other developer on the internet realized they need a list construct, and also would like to print statements in the language to a command prompt. Because I have written my implementation of the WHILE language in the final tagless style they are freely able to add these features to the core if they so desire. This scenario is demonstrated in the "Trivial Extensions, and Experiments" sections in the Extensions.hs file.

3 Challenges and Design Decisions

3.1 Using Final Tagless with Monads, A balancing act

For both the core and extended implementations of the WHILE language I make use of a state monad, holding a variable map as the state, and then instantiate the typeclasses using the state monad. The reader may have noticed that I awkwardly create a new data type like this to instantiate against:

```
-- | an eval monad
data EvalOne a = E (State (VarStore Prims) Prims)
```

Instead, I could do something like this, which is much cleaner:

```
-- | a cleaner eval monad
type EvalTwo a = State (VarStore Prims) Prims
```

Not only being more idiomatic haskell style, it maintains the phantom type variable and is explicit that I am really just creating a type synonym. However, if one tries to use this with the final tagless style, they will discover that it will not type check.

First, it will not type check because type synonyms *cannot be* partially applied for compiler reasons. Which means that everytime one uses a type synonym in haskell they must satisfy the kind of the type synonym. Normally this is not an issue because one rarely uses a type synonym to instantiate a type class. But in our case that is exactly what we are trying to do. Furthermore, the final tagless style *depends* on the constructor used in the instantiation to be of kind $\star \rightarrow \star$. EvalTwo is of kind $\star \rightarrow \star$, but can only be used in a type class instantiation as kind $\star$, because it cannot be partially applied. Thus, if we give the type constructor an "a" or anthing else in the instance declaration then it will be kind $\star$ (satisfying that the type synonym isn't partially applied), but that creates a type error for the type class because we are using constructor classes (which means the constructor must be kind $\star \rightarrow \star$). Thus, we can never create a class constructor for a type synonym because of this incongruity.

To get around this we are forced to create a new abstract data type to wrap the state monad. This is inconvienient because then we must constantly box/unbox the monad in order to use it, which is exactly what is observed in either the core or extended implementation.

Another restriction that I encountered mixing monads with the final tagless style is summed up in the following code:

```
var v = E e
  where e = do
          st <- get
          return $ st M.! v
```

Obviously this statement can be rewritten in a clean and elegant way using bind:

```
var v = get >>= E . return . (M.! v)
```

This form is much nicer and performs exactly the same, the problem is that it will not type check because we have wrapped our monad in a abstract data type yet have not made that type a monad instance. Thus, we will not type check and will incur errors saying there is not instance of MonadState for get, and Monad for return. While these instances could be added it is non-trivial (see below) and, I felt it was important to highlight that this is added boilerplate caused by the final tagless approach.

3.2 Lazy evaluation or the lack thereof

Consider the let statement implementation for the core language (it's identical to the extended version):

```
let_ v (E x) = E e
  where e = do
          st <- get
          x' <- x
          put $ M.insert v x' st
```

```
        return NoOp
```

Looking at this code snippet one might begin to wonder if instead of strictly evaluating "x"' to "x prime"', I could implement lazy evaluation by changing the call to M.insert from inserting "x prime"' to just inserting "x"'. The answer is no, because to do so would cause a type error in the Variable Store (whose type for values is "Prims"'). Basically we would be saving something of type "r a"' in the map instead of type "Prims"'. One may respond "We'll just change the type of the Variable Store!"'. The suggestion would look like this:

```
-- | Map to hold let bound variables
type VarStore a = M.Map String a

-- | an eval monad
data Eval a = E (State (VarStore (r a)) Prims)
```

Obviously this will not work. The first problem is that we have a type constructor "r" that is unbound, and therefore will cause a type error. But the only way to make a lazy evaluator for our final tagless WHILE language would be to add a statement, unevaluated, into the variable map. The only way to achieve the desired behavior is a definition like this:

```
-- | Map to hold let bound variables
type VarStore a = M.Map String a

-- | an eval monad
data TEval a = TE (State (VarStore (TEval Prims)) Prims)
```

But this forces one to make the data type an instance of MonadState. I'm not sure if it is possible to write this instance as all of my attempts ended in failure. Essentially the issue is a functional dependency in the MonadState typeclass, because of that dependency it becomes mandatory to describe how a value is returned from the data type. This is unclear to me; we would need to say, given a type TEval, we can uniquely determine (TEval Prims) from it. But to instantiate MonadState we need to provide definitions for the **get** and **put** functions. The **get** function does not take any arguments so we have no way of pattern matching the embedded monad, and similarly the **put** function only takes a state, but we still have no way of accessing the underlying monad. Even if pattern matching was possible it is not clear how, given the current definition of TEval, one would write this. I'm not suggesting this is impossible, but it is much harder to achieve than I originally thought.

Another design decision that encourages strict evaluation is the use of a homogeneous map. I chose to use a homogeneous map because it is closer to something I would consider safe in haskell, and because I wanted to stress test the tagless final approach. Heterogeneous maps do exist in haskell, but I consider them to not be in the spirit of haskell, and furthermore to be fairly exotic. Even if a heterogeneous map solves some of the problems stated here it would still, in my view, be a mark against the final tagless approach precisely because I needed to reach for such an exotic library. In any case, the use of a homogeneous map forces me to evaluate everything to a primitive before storing it, because the types of the values in the map must be the same, in this case the sum type Prims or NewPrims. If this were not the case then I would conceivably be able to implement the WHILE language without a the Prims datatype and thus any type in haskell could be fair game for use in the embedded language.

3.3 Adding Lists, one, at, a, time.

In the Extensions.hs file, in the experiments section I tried to add a polymorphic list to the WHILE language using patterns that worked for other types:

```
class Lists l where
  llit :: l a -> l [a]
  cons :: l a -> l [a] -> l [a]
  nth  :: l Int -> l [a] -> l a
  head :: l [a] -> l a
  tail :: l [a] -> l [a]
  map_ :: (a -> b) -> l [a] -> l [b]
  rightFold :: (a -> b -> b) -> l b -> l [a] -> l b
```

Similar to other types we start with a literal constructor, in this case it just takes something of type "a" and wraps it into a list and passes it to the "l" constructor to be part of the language. Almost all of the other functions follow in the same manner and have familiar types wrapped with the "l" type constructor. The problem is that this definition can never be instantiated, because we have no way to dispatch on the incoming "a" to select the right newPrim value constructor. In a tagged approach this definition would also be impossible. But it is important to note that in that approach the solution is more terse than the final tagless approach. First we would add the appropriate constructor to the language, this is also required in the final tagless approach:

```
data NewPrims = NI Int
              | NB Bool
              | S String -- extensions in constructor dimension
              | F Float
              | L [NewPrims]
              | Skip
```

Then when we write our semantics we would just pattern match on the list and handle it appropriately. In the final tagless approach we cannot pattern match. So instead we must make a list class that takes only the types we want. Essentially we want this:

```
class IntLists l where
  ilit  :: NewPrims -> l [NewPrims]
  icons :: NewPrims -> l [NewPrims] -> l [NewPrims]
  ihead :: l [NewPrims] -> l NewPrims
  itail :: l [NewPrims] -> l [NewPrims]
```

This works but my reservation is that it is not really in line with the final tagless approach. Basically it is leveraging the tagged primitives used in the language to add support for lists of primitives in the language. I'm not sure if this is actually "legal" in the final tagless style but I chose to avoid it. Rather then use, or abuse NewPrims as shown above I show how one can add lists to the language, and as an example I use integers. This approach more in the spirit of tagless final is shown in the extensions file but I'll present it here, as well:

```
class IntLists l a where
  ilit  :: a -> l [a]
  icons :: l a -> l [a] -> l [a]
  ihead :: l [a] -> l a
  itail :: l [a] -> l [a]
```

Basically we just abstract the type of the element into a type parameter "a". Then we can specify the type appropriately, dispatch on the right value constructor and maintain the spirt of the final tagless approach. The downside to this approach is that we will need to write more instances for each type that is legal in a list, effectively increasing our boilerplate count even more. Also note that this approach necessitates homogeneous lists, while the NewPrims driven one shown earlier can create heterogeneous lists.

Lastly, I want to mention that adding lists to the WHILE language in a final tagless approach necessitates that one cannot create hand-rolled loops for lists because we cannot pattern match. While this may not seem like a big problem it is relatively easy to imagine cases where hand-rolled loops may be easier to understand than nested folds and maps. Again, It is not impossible to do this in a final tagless way, its just an associated cost with the style that I wanted to highlight.

3.4 Attempts at adding Tagging, Goto/Gosub statements

Adding tagging, goto or gosub statements to the language suffers from a few complex matters. First, one could model a gosub statement (a statement that jumps to some labeled block) with a let statement whose body itself is a statement. On the evaluation of a gosub one would only need to pass the current state to the code block and then transfer control flow. This can be done, but because I have implemented strict evaluation it is basically impossible. If we could store statements with the let statement then it may be possible. A goto statement would be similar but take advantage of the line number instead of a named label, and then pass control flow to that line number. The issue with both of these is the vague notion of transferring control flow. How does one go about doing this in a tagless final approach? In the tagged approach we would recursively call our *eval* function on the statement returned

by the gosub lookup. For the goto we would need a list of linenumbers and statements, using that we could lookup the line number, truncate everything that should be skipped, map over the list to remove the line numbers and then recursively call eval on the resulting list of statements. In the tagless approach we have no way to recursively call our eval function because it does not exist. We could return the statement returned by the gosub lookup, or we could perform the same process described earlier with goto and then return the sequence of statements. Either way we don't actually evaluate to a primitive, and we have no way to do so.

Tagging statements suffers from similar problems. Typically one is able to tag an abstract syntax tree to simulate or determine line numbers in the tree. This is done by pattern matching and crawling down the tree transforming it to a list of line numbers and their associated statements. For the final tagless approach we have no way to pattern match on any statement, so traversing the AST becomes problematic because the AST is not just a series of nested function calls. Assuming that was solved, and we could traverse the tree, we would still have no way to crawl down sub-branches of say an if-statement. For an if-statment we would need to crawl down the then branch, retrieve the most recent line number, and then crawl down the else branch. However, we have no way of discerning an if-statement from any other statement, because we cannot pattern match. Again, this may be possible, but in the final tagless approach it is definitely not easily done.

4 Conclusion

In sum, the tagless final approach is quite clean but not without its own set of tradeoffs. In general I found that when I stayed close to simple extensions, like adding strings, floats and exponentials to the language it was quite easy. But even the slightest bit of complexity, like adding a list, led to an explosion in design concerns and considerations. My general take away is that the tagless final approach is very well suited to certain EDSLs, but once one starts running up against some of its rougher edges it becomes much more unwiedly. In the beginning of this project I was thinking something along the lines of "I'm a good functional programmer, I can survive without pattern matching", now, on the other side of the project, my view is basically that pattern matching isn't always essential, but having it is always desirable, and not being able to pattern match is a much greater cost than I originally thought.