Embedded Domain Specific Language Study – Tagless Final

Zixiu Su

2023-12-12

1 Introduction

As the final project for CSCI-P-424 Advanced Functional Programming class, this project is intended to study several techiniques to embed a **domain specific language** (DSL) into a typed host language (**Haskell**).

The main DSLs we choose are is a simply typed lambda calculus extended with integers. Our main goal is to study **tagless final**, and then compare it with other methods such as **free monad** and **GADTs**.

The methods are not exclusive – often, we can combine them. This study is only preliminary and empirical. But further investigation will be discussed.

For the most part, the project follows Oleg Kiselyov's tutorial on tagless final: `https://okmij.org/ftp/tagless-final/index.html`.

2 Naive Implementation of STLC in Haskell

As in `STLC/Naive.hs`, this approach mimics the CEK style interpreter taught in B521 Programming Languages Principle. But translation into a typed context complicates the situation.

The expressions of the language are defined as a datatype:

```
data Expr where
  Int :: Integer -> Expr
  Add :: Expr -> Expr -> Expr
  Var :: String -> Expr
  Lam :: String -> Expr -> Expr
  App :: Expr -> Expr -> Expr
```

The interpreter is attempted to be of type: `Expr -> Env -> Val`, so we define environment and value type as:

```
data Val where
  VInt  :: Integer -> Val
  VClos :: String -> Expr -> Env -> Val

type Env = Map String Val
```

Some expressions evaluate to integer values and some closures, the two have to be unionized under `Val` because the interpreter can only return one result type. This introduces **tags** (the constructors, `VInt` and `VClos`). Every time a value is used, it needs to be read off from `Val` by pattern matching, which brings boilerplate and run-time overhead.

For example, in the case of the expression `Add x y`:

```
interp (Add x y) env  = do
  xval <- interp x env
  yval <- interp y env
  xint <- unpack xval
  yint <- unpack yval
  return $ VInt (xint + yint)

unpack (VInt n) = Just n
unpack x        = error $ show x ++ " is not VInt"
```

Apparently, this is not an efficient way to embed STLC. The type does not contain enough information to describe the actual values. And Kiselyov gives a solution to the issue using GADTs.

3 GADTs Provide ”Tight” Embedding

As shown in `STLC/GADT.hs`, the expressions and variables are defined as:

```
data Var env t where
  V0 :: Var (t, env) t
  VS :: Var env      t -> Var (a, env) t

data Exp env t where
  B :: Bool                  -> Exp env Bool
  V :: Var env      t        -> Exp env t
```

```
L :: Exp (a, env) b        -> Exp env (a -> b)
A :: Exp env      (a -> b) -> Exp env a        -> Exp env b
```

Now the environments are represented by variable `env` in `Exp`, and it's tuple type which serves as a heterogenous container, which exposes the type of values.

Also, the type of the value of an expression is exposed by `t`, which allows the intepreter to return a (ad-hoc) polymorphic result:

```
interp :: Exp env t -> env -> t
interp (B b)    _   = b
interp (V var)  env = lookp var env
interp (L body) env = \arg -> interp body (arg, env)
interp (A f a)  env = (interp f env) (interp a env)

lookp :: Var env t -> env -> t
lookp (VS v) (_, xs) = lookp v xs
lookp V0     (x, _)  = x
```

This eliminates the "tags" in the union value type in the previous approach.

4 Tagless Final

Another way to achieve "tagless" is called tagless final. The duality is not so clear to the author in the formal, categorical sense. Some discussion can be found here: https://cstheory.stackexchange.com/questions/45565/what-category-are-tagless-final-algebras-final-in. The categorical semantics of tagless final requires further investigation, it might be related to CPS and Yoneda lemma.

The use of typeclass is a common theme among Haskell libraries, (eg.: `mtl`). In object-oriented programming, there is the similar concept of dependency injection. What justifies tagless final as a distinct concept seems also depends on the question in the previous paragraph.

4.1 Motivating Example

Starting from implementing a simple first order language (see `TriviaL.hs`), we follow the usual approach defining a recursive datatype `Exp` for the expressions, and then define the interpreter function by pattern matching. And, similarly, a pretty printer:

```
data Exp = Lit Int | Neg Exp | Add Exp Exp

interp :: Exp -> Int
interp (Lit x)   = x
interp (Neg x)   = -(interp x)
interp (Add x y) = interp x + interp y

pprint :: Exp -> String
pprint (Lit x)   = show x
pprint (Neg x)   = "-(" ++ pprint x ++ ")"
pprint (Add x y) = "(" ++ pprint x ++ "+" ++ pprint y ++ ")"
```

A family of functions can be implemented likewise, eg.: parser, typechecker, etc. However, using Haskell typeclass feature, we can treat the result types of these functions abstractly, and factor out the common interface:

```
class SymAdd repr where
  lit :: Int -> repr
  neg :: repr -> repr
  add :: repr -> repr -> repr
```

And implement each result type (`Int` for `interp`, `String` for `pprint`, etc.) as a concrete `repr`-esentation. For example, `interp :: Exp -> Int` becomes: (note `eval` only serves to choose the concrete representation/result type)

```
instance SymAdd Int where
  lit = id
  neg x = -x
  add x y = x + y

eval :: Int -> Int
eval = id
```

Programs now are "abstract" and up to concrete implementation. For example, we write 3+ (-4 + 5) as `prog0 = add (lit 3) (add (neg (lit 4)) (lit 5))`, which is typed as `prog0 :: SymAdd repr => repr`. Each instance of `SymAdd` can also be seens as a denotational semantics for the program.

4.2 STLC Using Tagless Final

As in `STLC/TaglessFinal.hs`, directly translating the expression definition in `STLC/GADT.hs` yields:

```
class Symantics repr where
  v0  :: repr (a, env) a
  vs  :: repr env a -> repr (any, env) a
  lam :: repr (a, env) b -> repr env (a -> b)
  app :: repr env (a -> b) -> repr env a -> repr env b

  int :: Integer -> repr env Integer
  add :: repr env Integer -> repr env Integer -> repr env Integer
```

For evaluation, we define the result type `R` using `newtype` and implement it as:

```
newtype R env a = R { unR :: env -> a }

instance Symantics R where
  v0   = R $ \(x, xs) -> x
  vs v = R $ \(_, xs) -> (unR v) xs

  lam body = R $ \xs -> \x -> (unR body) (x , xs)
  app f g  = R $ \xs -> (unR f) xs $ (unR g) xs

  int n   = R $ \xs -> n
  add a b = R $ \xs -> (unR a) xs + (unR b) xs
```

Notice the `R` and `unR` will be erased at run-time because the `newtype` wrapper does not change the same run-time representation, so they are different from the **tags** mentioned earlier.

Adopting a higher order abstract syntax (HOAS) would make the class definition even more simpler.

4.3 1# Register Machine

In order to test the tagless final method on more samples, we attempted to implement Larry Moss's **1#** register machine. Which is similar to Turing machines, for more details please see: https://iulg.sitehost.iu.edu/trm/.

To give a simplified intuition, the register machine contains two parts: 1. Program: a list of instructions and a program counter which points to the current instruction; 2. Registers: a list of locations, each contains some textual data

At each step, the machine carries out the current instruction until the program counter reaches the end of the program. The instructions define

actions such as: 1. shifting the program counter, 2. write symbols onto a particular register address, or 3. parse the first symbol of some address and shift the program counter based on the result.

The main discover from implementing 1# appeared when we tried to implement both instructions and machine states using typeclasses. Notice the machine's datatype is dependent on the instruction's because it contains instructions as data. And whenever a function is dependent on those abstract types as data, it is necessary to instantiate them into the inspectable "initial" representation (See `OneSharp/ThreeSharp.sh` line#98).

It remains an open question if the instantiation to the initial datatype is avoidable.

4.4 Typed Formatting

A formate template with typed holes can be used both for pretty printing and parsing. For example, `sprintf` and `sscanf` in **C**. The duality is also an interesting topic in bidirectional transformation.

As in `FPP.hs`, tagless final is particularly suitable here for typing the format template abstractly. Here are the typeclass definition and an example format with the inferred type:

```
class FormatSpec repr where
  lit  :: String -> repr a a
  int  :: repr a (Int -> a)
  char :: repr a (Char -> a)
  (^)  :: repr b c -> repr a b -> repr a c

-- fmt0 :: FormatSpec repr => repr a (Char -> a)
fmt0 = lit "Hello " ^ char
```

The type `repr a b` can be understood as result type `a` is extended by `b`. Components of the format template are concatenated by `(^)`. It is worth trying to derive the type of `fmt1 = char ^ int` by hand, which will illustrate the unification of type variables.

```
newtype FPr a b = FPr ((String -> a) -> b)

sprintf :: FPr String b -> b
sprintf (FPr format) = format id
```

```
instance FormatSpec FPr where
  lit str = FPr $ \k -> k str
  int     = FPr $ \k x -> k (show x)
  char    = FPr $ \k x -> k [x]
  (FPr a) ^ (FPr b)  =
    FPr $ \k -> a (\sa -> b (\sb -> k (sa ++ sb)))
```

The pretty printer representation is defind above as `FPr a b`. Observe its type is almost a CPS monad, but with two result types. The parser is defined dually, and it would be a great exercise fleshing it out using the hints given.

4.5 Summary

The benefits of adopting tagless final are:

1. There is no dispatch overhead on pattern matching of the expression. The abstract expression does the recursive calls automatically and combine their results, akin to a recursive schemes. Which brings performance improvements.

2. It is easier to achieve incremental development, compared to adding cases in the AST datatype and modifying each function mapping out from that datatype. Adding new terms is done by defining new typeclasses. Old programs will not be broken this way. New programs will contain more constraints in their type signatures. And presumably there is no need to recompile previous code (need to verify).

Some directions to extend the current project:

1. Extend STLC with `callCC`.

2. Solve the problem posed by **1#** (avoid using initial datatypes).

3. Study the semantics of tagless final and its polymorphism.

5 Free Monads

The project was intended to compare tagless final with free monads, but it turns out the two approaches are somewhat orthogonal. A free monad is indeed an initial object, we might as well ”finalize” it – define the free monad as a typeclass and implement concrete interpretations as instances.

Unfortunately due to time limit, the free monad approach could not be investigated properly, even though it has a clear categorical semantics. Here we briefly report the "user experience" as we learn to implement simple programs with free monads. The calculator example is modified from examples in Nikolay Yakimov's tutorial: https://serokell.io/blog/introduction-to-free-monads. And we expanded it to STLC, thanks to the help of Artem Yurchenko and his example project. All code related to free monads resides in `FreeMonad.hs`.

5.1 Library Functions

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

The `Free` datatype lifts a functor `f` to its free monad. This provides a uniform construction for various of monads, eg. `State`, `List`, `Writer`, etc., from their underlying functor, without writing each of their applicative and monad instances. Instead we only need to define the instances on `Free f`, and then write an interpreter to do actual computation, eg:

```
type State s a = s -> (a, s) -- `(State s)` is a functor on `a`
runState :: Free (State s) a -> State s a
runState (Pure a) = \s -> (a, s)
runState (Impure fm) = \s -> let (m, s') = g s in runState m s'
```

To interpret a free monad to other monads, we have the `foldFree` function, which is essentially lifts an one-step functor evaluation to a recursive free monad evaluation:

```
foldFree :: (Functor f, Monad m) =>
    (forall x. f x -> m x) -> Free f a -> m a
foldFree _   (Pure a) = return a
foldFree phi (Impure ffa) =
  join $ phi (fmap (foldFree phi) ffa)
```

5.2 A Calculator

Similar to the trivial first order language before, the calculator DSL only has integers, addition, and I/O. The underlying functor is defined as: (note the continuations `(X -> r)` presumably can be eliminated using **freer monads**)

```
data CalF t r where
  Lit    :: Int -> (Int -> r) -> CalF t r
  Plus   :: Int -> Int -> (Int -> r) -> CalF t r
  Input  :: (t -> r) -> CalF t r
  Output :: t -> (() -> r) -> CalF t r
```

Then the following term constructors are defined. This boilerplate should be automated using `makeFree` in the Haskell library: https://hackage.haskell.org/package/free-5.2/docs/Control-Monad-Free-TH.html.

```
lit :: Int -> Cal Int Int
lit x = liftF $ Lit x id

plus :: Int -> Int -> Cal Int Int
plus x y = liftF $ Plus x y id

input :: Cal t t
input = liftF $ Input id

output :: t -> Cal t ()
output x = liftF $ Output x id

prog1 :: Cal Int ()
prog1 = do
  a <- lit (-1)
  x <- input
  r <- plus x a
  output r
```

The program now is constructed in a monadic style, which provides some syntactic convenience. However, this forces the programmer to choose the evaluation order and linearize the expression AST. But sometimes we want to have access to the tree representation, eg. pretty printing. This might be a separate concern.

We can modify the underlying functor to make it possible, by replacing all `t`'s with `Cal t t`'s in `CalF` definition (subtrees become free monads instead of unlifted values, analogous to the LC example at line#176). This mutual recursion complicates the evaluation because the recursive function has to be called on subtrees when defining the one step evaluation (see line#229).

```
calculateF :: CalF Int x -> IO x
```

```
calculateF (Lit x k)    = return $ k x
calculateF (Plus x y k) = return $ k (x + y)
calculateF (Input k)    = do { x <- read <$> getLine ; return $ k x }
calculateF (Output x k) = do { print x ; return $ k ()  }

calculate :: Cal Int x -> IO x
calculate = foldFree calculateF
```

The interpreter is written as above. Our STLC is implemented likewise, we omit the code here since it's not noteworthy. The main question here is whether to use the monadic program format to replace a tree like

A few things to work on further:

1. Use freer monads
2. Discuss the relation between algebras and monads
3. Investigate the efficiency issue of using free monads