Monad Transformers Step by Step

Kevin Macwan - 163050053

November 23, 2016

1 Abstract

In this tutorial, we describe how to use monad transformers in order to incrementally add functionality to Haskell programs. It is not a paper about implementing transformers, but about using them to write elegant, clean and powerful programs in Haskell. Starting from an evaluation function for simple expressions, we convert it to monadic style and incrementally add error handling, environment passing, state and logging by composing monad transformers.

2 Introduction

Monads are a remarkably elegant way for structuring programs in a flexible and extensible way. They are especially interesting in a lazy functional language like Haskell, because they allow the integration of side-effects into otherwise purely functional programs. Furthermore, by structuring a program with monads, it is possible to hide much of the necessary book-keeping and plumbing necessary for many algorithms in a handful of definitions specific for the monad in use, removing the clutter from the main algorithm.
Monad transformers offer an additional benefit to monadic programming: by providing a library of different monads and types and functions for combining these monads, it is possible to create custom monads simply by composing the necessary monad transformers. For example, if you need a monad with state and error handling, just take the StateT and ErrorT monad transformers and combine them. The goal of this paper is to give a gentle introduction to the use of monad transformers by starting with a simple function and extending it step by step with various monadic operations in order to extend its functionality. This paper is not about the theory underlying the monad transformer concept, and not about their implementation (except for what is necessary for successfully using them).

2.1 Sample Example

Below is our simple programming language:

```
import Control.Monad.Identity
import Control.Monad.Error
import Control.Monad.Reader
import Control.Monad.State
import Control.Monad.Writer
import Data.Maybe
import qualified Data.Map as Map

type Name = String                          -- variable names
data Exp = Lit Integer                      -- expressions
           |Var Name
           |Plus Exp Exp
           |Abs Name Exp
           |App Exp Exp
   deriving(Show)

data Value = IntVal Integer                 -- values
             |FunVal Env Name Exp
    deriving(Show)

type Env = Map.Map Name Value               -- mapping from names to values
```

Here is the interpreter for the above language:

```
eval0 :: Env -> Exp -> Value
eval0 env (Lit i) = IntVal i
eval0 env (Var n) = fromJust (Map.lookup n env)  -- Map.lookup returns Maybe Value
eval0 env (Plus e1 e2) = let IntVal i1 = eval0 env e1
                             IntVal i2 = eval0 env e2
                         in IntVal (i1+i2)
eval0 env (Abs n e) = FunVal env n e
eval0 env (App e1 e2) = let val1 = eval0 env e1
                            val2 = eval0 env e2
                        in case val1 of
                             FunVal env' n body -> eval0 (Map.insert n val2 env') body
```

$12 + ((\lambda x \; x)(4 + 2))$ can be represented as :
exampleExp = Lit 12 'Plus' (App (Abs "x" (Var "x")) (Lit 4 'Plus' Lit 2))

eval0 Map.empty exampleExp = IntVal 18

3 Monad Transformers

3.1 converting to Monadic Style using Identity Monad

we will use Identity monad as a "base case", around which other monad transformers can be wrapped.

```
eval1 :: Env -> Exp -> Identity Value
eval1 env (Lit i) = return (IntVal i)
eval1 env (Var n) = return (fromJust (Map.lookup n env))
eval1 env (Plus e1 e2) = do IntVal i1 <- eval1 env e1
                            IntVal i2 <- eval1 env e2
                            return (IntVal (i1+i2))
eval1 env (Abs n e) = return (FunVal env n e)
eval1 env (App e1 e2) = do val1 <- eval1 env e1
                           val2 <- eval1 env e2
                           case val1 of
                             FunVal env' n body -> eval1 (Map.insert n val2 env') body
```

runIdentity (eval1 Map.empty exampleExp) = IntVal 18

Note: The type of eval1 could be generalized to
eval1 :: Monad m $\Rightarrow$ Env $\rightarrow$ Exp $\rightarrow$ m Value

3.2 Adding Error Handling

```
eval2 :: Env -> Exp -> ErrorT String Identity Value
eval2 env (Lit i) = return (IntVal i)
eval2 env (Var n) = case Map.lookup n env of
                     Nothing -> throwError ("unbound variable: "++n)
                     Just val -> return val
eval2 env (Plus e1 e2) = do e1' <- eval2 env e1
                            e2' <- eval2 env e2
                            case (e1',e2') of
                              (IntVal i1,IntVal i2) -> return (IntVal (i1+i2))
                              _ -> throwError "type error in addition"
eval2 env (Abs n e) = return (FunVal env n e)
eval2 env (App e1 e2) = do val1 <- eval2 env e1
                           val2 <- eval2 env e2
                           case val1 of
                FunVal env' n body -> eval2 (Map.insert n val2 env') body
                _ -> throwError "type error in application"
```

```
runIdentity
   (runErrorT (eval2 Map.empty exampleExp))              => (IntVal 18)
runIdentity
   (runErrorT (eval2 Map.empty
                    (Plus (Lit 1) (Abs "x" (Var "x"))))) => Left "type error in addition"
runIdentity
   (runErrorT (eval2 Map.empty (Var "x")))               => Left "unbound variable: x"
```

The function for running a computation in the Eval2 monad changes in two ways. First, the result of evaluation is now of type Either String Value, where the result Left s indicates that an error has occurred with error message s, or Right r , which stands for successful evaluation with result r . Second, we need to call the function runErrorT on the given computation to yield an Identity computation, which can in turn be evaluated using runIdentity.

3.3 Hiding environment using Reader Monad

One way to make the definition of the evaluation function even more pleasing is to hide the environment from all function definitions and calls. Since there is only one place where the environment is extended (for function application) and two places where it is actually used (for variables and expressions), we can reduce the amount of code by hiding it in all other places. This will be done by adding a **ReaderT** monad transformer in order to implement a reader monad. A reader monad passes a value into a computation and all its sub-computations.
In Haskell the Reader Monad is the standard way to handle functions which need to read some sort of immutable environment.

```
eval3 :: Exp ->  ReaderT Env (ErrorT String Identity) Value
eval3 (Lit i) = return (IntVal i)
eval3 (Var n) = do env <- ask
                   case Map.lookup n env of
                     Nothing -> throwError ("unbound variable: "++n)
                     Just val -> return val
eval3 (Plus e1 e2) = do e1' <- eval3 e1
                        e2' <- eval3 e2
                        case (e1',e2') of
                            (IntVal i1,IntVal i2) -> return (IntVal (i1+i2))
                            _ -> throwError "type error in addition"
eval3 (Abs n e) = do env <- ask
                     return (FunVal env n e)
eval3 (App e1 e2) = do val1 <- eval3 e1
                       val2 <- eval3 e2
                       case val1 of
                          FunVal env' n body ->
                                                 local (const (Map.insert n val2 env'))
                                                 (eval3 body)
                          _ -> throwError "type error in application"


runIdentity (runErrorT
             (runReaderT (eval3 exampleExp) Map.empty)) => Right (IntVal 18)
```

In all places where the current environment is needed, it is extracted from the hidden state of the reader monad using the **ask** function. In the case of function application, the local function is used for modifying the environment for the recursive call. **Local** has the type (r $\rightarrow$ r) $\rightarrow$ m a $\rightarrow$ m a, that is we need to pass in a function which maps the current environment to the one to be used in the nested computation, which is the second argument. In our case, the nested environment does not depend on the current environment, so we simply pass in a constant function using **const**.

The Reader Monad works within the context of a shared environment. But what does that mean? Say you needed some shared object to execute a bunch of functions. An example could be that you need a database connection in every query function you execute. Or it could be some configuration options read from a file that are needed across a number of functions.

3.4 Adding State

Another important application of monads is to provide mutable state to otherwise purely functional code. This can be done using a State monad, which provides operations for specifying an initial state, querying the current state and changing it.
Suppose if we want to add profiling capabilities to our little interpreter,we can define new monad by wraping StateT constructor monad around the innermost monad,Identity.
Suppose we want ot count number of evaluation steps that is the number of calls to our eval function.

tick function will take state and increment value by one and will put it back.

```
tick = do st <- get
          put (st+1)
eval4 :: Exp -> ReaderT Env (ErrorT String (StateT Integer Identity)) Value
eval4 (Lit i) = do tick
                   return $IntVal i
eval4 (Var n) = do
                 tick
                 env <- ask
                 case Map.lookup n env of
                    Nothing ->throwError ("unbound variable: "++n)
                    Just val-> return val
eval4 (Plus e1 e2) = do
                      tick
                      e1' <- eval4 e1
                      e2' <- eval4 e2
                      case (e1',e2') of
                        (IntVal i1,IntVal i2) -> return $IntVal (i1 +i2)
                         _->throwError "type error in addition"
eval4 (Abs n e) = do
                   tick
                   env <- ask
                   return $FunVal env n e
eval4 (App e1 e2) = do
                     tick
                     val1 <- eval4 e1
                     val2 <- eval4 e2
                     case val1 of
                       FunVal env' n body ->local
                                            (const (Map.insert n val2 env')) (eval4 body)
                       _->throwError "type error in application"

runIdentity (runStateT
              (runErrorT
                (runReaderT (eval4 exampleExp) Map.empty)) 0)  => (Right (IntVal 18),8)
```

The return type of the function changes, because the final state is returned together with the evaluation result.

If we swap **ErrorT** and **StateT** with each other then result will differ from previous computation. First evaluation will be carried out which will generate some value or error.That will be passed to StateT next. In case of Normal value it will work the same but if error was generate than state monad will also pass error back. Whereas in previous case we were getting tuple - Result and State;Result can be valid value or error-in both cases we were getting state.

The position of the reader monad transformer does not matter, since it does not contribute to the final result.

3.5 Adding Logging using WriterT

WriterT is in some sense dual to ReaderT , because the functions it provides let you add values to the result of the computation instead of using some values passed in.

In the evaluation function, we illustrate the use of the writer monad by writing out the name of each variable encountered during evaluation.

```
eval5 :: Exp -> ReaderT Env (ErrorT String
                              (WriterT [String] (StateT Integer Identity))) Value
eval5 (Lit i) = do tick
                   return $IntVal i
eval5 (Var n) = do tick
                   tell [n]
                   env <- ask
                   case Map.lookup n env of
                     Nothing ->throwError ("unbound variable: "++n)
                     Just val-> return val
eval5 (Plus e1 e2) = do tick
                        e1' <- eval5 e1
                        e2' <- eval5 e2
                        case (e1',e2') of
                          (IntVal i1,IntVal i2) -> return $IntVal (i1 +i2)
                           _->throwError "type error in addition"
eval5 (Abs n e) = do tick
                     env <- ask
                     return $FunVal env n e
eval5 (App e1 e2) = do tick
                       val1 <- eval5 e1
                       val2 <- eval5 e2
                       case val1 of
                         FunVal env' n body ->
                                 local (const (Map.insert n val2 env')) (eval5 body)
                         _->throwError "type error in application"
```

4 Conclusion

Monad transformers are a powerful tool in the toolbox of a functional programmer. This paper introduces several of the monad transformers available in current Haskell implementations, and shows how to use and combine them in the context of a simple functional interpreter.
The use of monad transformers makes it very easy to define specialized monads for many applications, reducing the temptation to put everything possibly needed into the one and only monad hand-made for the current application.

5 References

1. Monad Transformers Step by Step - Martin Grabmuller