Free monads

Haskell and Cryptocurrencies

Dr. Andres Löh, Well-Typed LLP
Dr. Lars Brünjes, IOHK
Dr. Polina Vinogradova, IOHK

2019-02-18

INPUT | OUTPUT

Goals

- Case study monad transformers
- Free monads

Case study: Evolving an interpreter

A simple language for expressions

```
data Expr =
    Lit Int
  | Add Expr Expr
```

A simple language for expressions

```haskell
data Expr =
     Lit Int
   | Add Expr Expr
```

```haskell
eval :: Expr -> Int
eval (Lit n)     = n
eval (Add e1 e2) = eval e1 + eval e2
```

Adding division

```haskell
data Expr =
     Lit Int
   | Add Expr Expr
   | Div Expr Expr
```

Adding division

```haskell
data Expr =
    Lit Int
  | Add Expr Expr
  | Div Expr Expr
```

```haskell
eval :: Expr -> Either String Int
eval (Lit n)     = pure n
eval (Add e1 e2) = (+) <$> eval e1 <*> eval e2
eval (Div e1 e2) = do
  v1 <- eval e1
  v2 <- eval e2
  if v2 == 0
    then throwError "division by 0"
    else return (v1 `div` v2)
```

Two evaluator versions

```haskell
eval :: Expr -> Int
eval (Lit n)      = n
eval (Add e1 e2) = eval e1 + eval e2
```

```haskell
eval :: Expr -> Either String Int
eval (Lit n)      = pure n
eval (Add e1 e2) = (+) <$> eval e1 <*> eval e2
eval (Div e1 e2) = do
  v1 <- eval e1
  v2 <- eval e2
  if v2 == 0
    then throwError "division by 0"
    else return (v1 `div` v2)
```

Rewrite affects every line.

Rewrite to identity monad

```haskell
eval :: Expr -> Identity Int
eval (Lit n)       = pure n
eval (Add e1 e2) = (+) <$> eval e1 <*> eval e2
```

```haskell
eval :: Expr -> ExceptT String Identity Int
eval (Lit n)       = pure n
eval (Add e1 e2) = (+) <$> eval e1 <*> eval e2
eval (Div e1 e2) = do
  v1 <- eval e1
  v2 <- eval e2
  if v2 == 0
    then throwError "division by 0"
    else return (v1 `div` v2)
```

Rewrite now affects only the new case.

Intermediate summary and plan

- If we accept applicative / monadic style, we seem to be able to add new effects to our evaluator with minimal impact on existing cases.
- Let's define an "abstract" monad `Eval` that we gradually equip with new features.
- We will use monad transformers to implement the interface dictated by `Eval`.
- Extensions typically require adapting only the interesting cases of the evaluator, and possibly the monad.

The `Eval` monad

Required interface:

```
data Eval   -- abstract

divByZeroError :: Eval a
runEval        :: Eval a -> Either String a
```

Evaluator with division, again

```haskell
eval :: Expr -> Eval Int
eval (Lit n)      = pure n
eval (Add e1 e2) = (+) <$> eval e1 <*> eval e2
eval (Div e1 e2) = do
  v1 <- eval e1
  v2 <- eval e2
  if v2 == 0
    then divByZeroError
    else return (v1 `div` v2)
```

Implementing the interface

```
newtype Eval a =
  Eval (ExceptT String Identity a)
  deriving
    ( Functor, Applicative
    , Monad, MonadError String
    )
```

- A `newtype` makes it easier to still treat `Eval` as abstract for most of the code.
- The `GeneralizedNewtypeDeriving` extension makes it possible to lift class instances from the inner type to the wrapper type in the obvious way.

Implementing the interface (contd.)

```haskell
divByZeroError :: Eval a
divByZeroError = throwError "division by 0"
```

```haskell
runEval :: Eval a -> Either String a
runEval (Eval m) =
  runIdentity (runExceptT m)
```

Adding variables

```haskell
data Expr =
    Lit Int
  | Add Expr Expr
  | Div Expr Expr
  | Var String
```

Adding variables

```haskell
data Expr =
    Lit Int
  | Add Expr Expr
  | Div Expr Expr
  | Var String
```

`Eval` interface extension / modification:

```haskell
varLookup :: String -> Eval Int
runEval   :: Eval a -> Env -> Either String a
```

Adapting the evaluator

```haskell
eval :: Expr -> Eval Int
eval (Lit n)       = pure n
eval (Add e1 e2) = (+) <$> eval e1 <*> eval e2
eval (Div e1 e2) = do
  v1 <- eval e1
  v2 <- eval e2
  if v2 == 0
    then divByZeroError
    else return (v1 `div` v2)
eval (Var x)       = varLookup x
```

Only the last case is new; all others unchanged.

Implementing the interface extension

```haskell
newtype Eval a =
  Eval (ReaderT Env (ExceptT String Identity) a)
  deriving
    ( Functor, Applicative
    , Monad, MonadError String, MonadReader Env)
```

Monad changes. Now we include a reader.

Implementing the interface extension (contd.)

```
varLookup :: String -> Eval Int
varLookup x = do
  env <- ask
  case M.lookup x env of
    Nothing -> unknownVar x
    Just n   -> return n
```

```
unknownVar :: String -> Eval a
unknownVar x =
  throwError $ "unknown: " ++ show x
```

Implementing the interface extension (contd.)

```
runEval :: Eval a -> Env -> Either String a
runEval (Eval m) env =
  runIdentity (runExceptT
    (runReaderT m env))
```

The code for `divByZeroError` needs no changes.

Examples

```
GHCi> runEval (eval (Div (Var "x") (Var "x")))
        Map.singleton "x" 2
Right 1

GHCi> runEval (eval (Div (Var "x") (Var "x")))
        Map.singleton "x" 0
Left "division by 0"

GHCi> runEval (eval (Div (Var "x") (Var "x")))
        Map.empty
Left "unknown: \"x\""
```

Adding mutation

```haskell
data Expr =
    Lit Int
  | Add Expr Expr
  | Div Expr Expr
  | Var String
  -- these are new:
  | Seq Expr Expr
  | Assign String Expr
```

Adding mutation

```haskell
data Expr =
     Lit Int
   | Add Expr Expr
   | Div Expr Expr
   | Var String
  -- these are new:
   | Seq Expr Expr
   | Assign String Expr
```

`Eval` interface extension:

```haskell
varSet :: String -> Int -> Eval ()
```

Adapting the evaluator

Again, all the old cases are unchanged:

```
eval :: Expr -> Eval Int
eval (Seq e1 e2)  = eval e1 >> eval e2
eval (Assign x e) = do
  v <- eval e
  varSet x v
  return v
...
```

Implementing the interface extension

```haskell
newtype Eval a =
  Eval (StateT Env (ExceptT String Identity) a)
  deriving
    ( Functor, Applicative
    , Monad, MonadError String, MonadState Env)
```

We are changing `MonadReader` to `MonadState`.

Implementing the interface extension (contd.)

```
varSet :: String -> Int -> Eval ()
varSet x v =
  modify (M.insert x v)
```

Recall:

```
modify :: MonadState s m => (s -> s) -> m ()
modify f = do
  s <- get
  put (f s)
```

Implementing the interface extension (contd.)

Minor changes in `varLookup` and `runEval`:

```
varLookup :: String -> Eval Int
varLookup x = do
  env <- get
  case M.lookup x env of
    Nothing -> throwError $ "unknown: " ++ show x
    Just n  -> return n
```

```
runEval :: Eval a -> Env -> Either String a
runEval (Eval m) env =
  runIdentity (runExceptT
    (evalStateT m env))
```

Examples

```haskell
program :: Expr
program =
  Assign "x" (Lit 16)
   `Seq` Assign "x" (Div (Var "x") (Lit 2))
   `Seq` Add (Var "x") (Lit 1)
```

```
GHCi> runEval (eval program) Map.empty
Right 9
```

Summary

- Using an abstract monad, we can easily extend code with new kinds of effects, usually without affecting existing code.
- Monad transformers are a useful way to quickly implement such interfaces.
- The notion of sequencing changes with changing the underlying monad. We have a “programmable semicolon”.

Revisiting the interpreter

(Extended) Expression language

```haskell
data Expr =
      Lit Int
    | Add Expr Expr
    | Div Expr Expr
    | Var String
    | Seq Expr Expr
    | Assign String Expr
```

Evaluator

```haskell
eval :: Expr -> Eval Int
eval (Div e1 e2) = do
  v1 <- eval e1
  v2 <- eval e2
  if v2 == 0
    then divByZeroError
    else return (v1 `div` v2)
...
```

Evaluation monad

```haskell
data Eval   -- abstract

instance Monad Eval

divByZeroError :: Eval a
unknownVar     :: String -> Eval a
varLookup      :: String -> Eval Int
varSet         :: String -> Int -> Eval ()

runEval        :: Eval a -> Env -> Either String a
```

Evaluation monad

```haskell
data Eval  -- abstract

instance Monad Eval

divByZeroError :: Eval a
unknownVar     :: String -> Eval a
varLookup      :: String -> Eval Int
varSet         :: String -> Int -> Eval ()

runEval        :: Eval a -> Env -> Either String a
```

- Had an implementation using monad transformers.
- Can we implement in a way that abstracts from the interface?

Abstracting from the interface

- Provide several different implementations of the interface.
- Have programs in the expression language available as data.
- Analyze programs written in the expression language and possibly transform or even compile them.

Abstracting from the interface

- Provide several different implementations of the interface.
- Have programs in the expression language available as data.
- Analyze programs written in the expression language and possibly transform or even compile them.

Basic idea: turn functions into constructors.

A first attempt

```haskell
data Eval a =
    DivByZeroError
  | UnknownVar String
  | VarLookup String
  | VarSet String Int
```

A first attempt

```haskell
data Eval a =
    DivByZeroError
  | UnknownVar String
  | VarLookup String
  | VarSet String Int
```

This loses info about the result types:

```haskell
VarLookup :: String -> Eval a
```

vs.

```haskell
varLookup :: String -> Eval Int
```

Also, it’s unclear how to define a monad instance.

Generalized Algebraic Datatypes (GADTs)

Using a GADT for `Eval`

```
data Eval :: * -> * where
  DivByZeroError :: Eval a
  UnknownVar     :: String -> Eval a
  VarLookup      :: String -> Eval Int
  VarSet         :: String -> Int -> Eval ()
```

- Constructors are listed by type signature.
- We can restrict the result type.

Using a GADT for `Eval`

```
data Eval :: * -> * where
  DivByZeroError :: Eval a
  UnknownVar     :: String -> Eval a
  VarLookup      :: String -> Eval Int
  VarSet         :: String -> Int -> Eval ()
  Return         :: a -> Eval a
  Bind           :: Eval a -> (a -> Eval b)
                      -> Eval b
```

- Constructors are listed by type signature.
- We can restrict the result type.
- We can do the same for the monad operations.

Instances

```haskell
instance Monad Eval where
  return = Return
  (>>=)  = Bind
instance Functor Eval where
  fmap   = liftM
instance Applicative Eval where
  pure   = return
  (<*>)  = ap
```

Wrappers

To literally implement the interface, we need wrappers:

```
divByZeroError = DivByZeroError
unknownVar     = UnknownVar
varLookup      = VarLookup
varSet         = VarSet
```

Interpreting the GADT

Let `MT` be the module defining the evaluation monad using monad transformers.

```
fromEval :: Eval a -> MT.Eval a
fromEval DivByZeroError = MT.divByZeroError
fromEval (UnknownVar x) = MT.unknownVar x
fromEval (VarLookup x)  = MT.varLookup x
fromEval (VarSet x v)   = MT.varSet x v
fromEval (Return x)     = return x
fromEval (Bind m f)     =
  fromEval m >>= fromEval . f
```

Monad laws

```
return x >>= f = f x
a >>= return   = a
(a >>= f) >>= g = a >>= (\ x -> f x >>= g)
```

Monad laws

```
return x >>= f = f x
a >>= return   = a
(a >>= f) >>= g = a >>= (\ x -> f x >>= g)
```

Does the `Eval` GADT adhere to the monad laws?

Monad laws

```
return x >>= f = f x
a >>= return   = a
(a >>= f) >>= g = a >>= (\ x -> f x >>= g)
```

Does the `Eval` GADT adhere to the monad laws?

Strictly speaking, it quite obviously does not.

How bad is this?

- One can argue that violating the monad laws to a certain extent is ok if the violation is ultimately not observable.
- For example, if `Eval` is always interpreted into `MT.Eval` and `MT.Eval` adheres to the monad laws, that seems ok.
- But for every function we define that operates on `Eval`, we inherit a proof obligation that it must be compatible with the monad laws.
- Wouldn't it be nice if that wasn't necessary, and we could just define a variant of `Eval` that *does* adhere to the monad laws?

Free monads

Observation

In essence, the monad laws say that every monadic computation has a normal form:

```
do
  x1 <- step1
  x2 <- step2
  ...
  xn <- stepn
  return something
```

Observation

In essence, the monad laws say that every monadic computation has a normal form:

```
do
  x1 <- step1
  x2 <- step2
   ...
  xn <- stepn
  return something
```

- Every individual step is followed by a bind.
- We don't need `Bind` as a constructor if we pair it with steps.

Normalising steps

```haskell
data Eval :: * -> * where
  DivByZeroError :: Eval a
  UnknownVar     :: String -> Eval a
  VarLookup      :: String -> Eval Int
  VarSet         :: String -> Int -> Eval ()
  Return         :: a -> Eval a
  Bind           :: Eval a -> (a -> Eval b)
                        -> Eval b
```

Normalising steps

```
data Eval :: * -> * where
  DivByZeroError :: Eval a
  UnknownVar     :: String -> Eval a
  VarLookup      :: String -> Eval Int
  VarSet         :: String -> Int -> Eval ()
  Return         :: a -> Eval a
  Bind           :: Eval a -> (a -> Eval b)
                      -> Eval b

varLookup :: String -> (Int -> Eval a) -> Eval a
varLookup x = Bind (VarLookup x)

varSet :: String -> Int -> Eval a -> Eval a
varSet x v k = Bind (VarSet x v) (const k)
```

Normalising steps

```
data Eval :: * -> * where
  DivByZeroError :: Eval a
  UnknownVar     :: String -> Eval a


  Return         :: a -> Eval a



  VarLookup      :: String -> (Int -> Eval a)
                        -> Eval a
  VarSet         :: String -> Int -> Eval a
                        -> Eval a
```

Normalising steps

```haskell
data Eval :: * -> * where
  DivByZeroError :: Eval a
  UnknownVar     :: String -> Eval a
  VarLookup      :: String -> (Int -> Eval a)
                       -> Eval a
  VarSet         :: String -> Int -> Eval a
                       -> Eval a
  Return         :: a -> Eval a
```

- No longer a proper GADT.
- We could (but don't need to) add continuations to the error cases, because they "never return" and ignore their continuations.

Normalising steps

```haskell
data Eval a =
     DivByZeroError
   | UnknownVar String
   | VarLookup String (Int -> Eval a)
   | VarSet String Int (Eval a)
   | Return a
```

Monad instance

```haskell
instance Monad Eval where
  return = Return

  Return x         >>= f = f x
  DivByZeroError   >>= _ = DivByZeroError
  UnknownVar x     >>= _ = UnknownVar x
  VarLookup x k    >>= f = VarLookup x
                              (\ v -> k v >>= f)
  VarSet x v k     >>= f = VarSet x v (k >>= f)
```

Instances for `Functor` and `Applicative` as usual.

Bind is substitution

If we look at terms of type `Eval a` as trees with `Return` at the leaves, then `(>>=)` implements substitution.

Still implementing the interface?

```haskell
divByZeroError :: Eval a
divByZeroError = DivByZeroError

unknownVar :: String -> Eval a
unknownVar = UnknownVar

varLookup :: String -> Eval Int
varLookup x = VarLookup x Return

varSet :: String -> Int -> Eval ()
varSet x v = VarSet x v (Return ())
```

Still possible to write an interpreter?

```
fromEval :: Eval a -> MT.Eval a
fromEval DivByZeroError  = MT.divByZeroError
fromEval (UnknownVar x)  = MT.unknownVar x
fromEval (VarLookup x k) =
  MT.varLookup x >>= fromEval . k
fromEval (VarSet x v k)  =
  MT.varSet x v >> fromEval k
fromEval (Return x)      = return x
```

And the monad laws hold as well!

For example,

```
return x >>= f = f x
```

holds by construction.

And the monad laws hold as well!

For example,

```
return x >>= f = f x
```

holds by construction.

The other two laws can be proved via a simple induction.

Another similar example

Recall the exercises:

```
data GP a =
    End a
  | Get (Int -> GP a)
  | Put Int (GP a)
```

Another similar example

Recall the exercises:

```haskell
data GP a =
    End a
  | Get (Int -> GP a)
  | Put Int (GP a)
```

```haskell
instance Monad GP where
  return = End

  End a     >>= f = f a
  Get k     >>= f = Get (\ n -> k n >>= f)
  Put n k >>= f = Put n (k >>= f)
```

Again, `(>>=)` implements substitution.

Free monad generalization

Abstracting from the steps

```
data Eval a =
    DivByZeroError
  | UnknownVar String
  | VarLookup String (Int -> Eval a)
  | VarSet String Int (Eval a)
  | Return a
```

Abstracting from the steps

```
data EvalOp a =
    DivByZeroError
  | UnknownVar String
  | VarLookup String (Int -> Eval a)
  | VarSet String Int (Eval a)

data Eval a =
    Return a
  | Wrap (EvalOp a)
```

Abstracting from the steps

```
data EvalOp b =
    DivByZeroError
  | UnknownVar String
  | VarLookup String (Int -> b)
  | VarSet String Int b

data Eval a =
    Return a
  | Wrap (EvalOp (Eval a))
```

Abstracting from the steps

```
data EvalOp b =
    DivByZeroError
  | UnknownVar String
  | VarLookup String (Int -> b)
  | VarSet String Int b

data Free f a =
    Return a
  | Wrap (f (Free f a))

type Eval = Free EvalOp
```

`Free` vs. `Fix`

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

```haskell
data Free f a =
     Return a
   | Wrap (f (Free f a))
```

The `Free` type is like a type-level fixed point with a built-in possibility to stop via `Return`.

The same construction for `GP`

```haskell
data GP a =
    End a
  | Get (Int -> GP a)
  | Put Int (GP a)
```

```haskell
data GPOp b =
    Get (Int -> b)
  | Put Int b

type GP = Free GPOp
```

Monad instance for `Free`

```haskell
data Free f a =
    Return a
  | Wrap (f (Free f a))
```

```haskell
instance Functor f => Monad (Free f) where
  return :: a -> Free f a
  return = Return
  (>>=) :: Free f a -> (a -> Free f b) -> Free f b
  Return x >>= f = f x
  Wrap c   >>= f = Wrap (fmap (>>= f) c)
```

For every functor, we get a monad instance for free.

Free structures, categorically

In category theory, a structure is called free if it makes no more assumptions as necessary.

Free structures, categorically

In category theory, a structure is called free if it makes no more assumptions as necessary.

If we start with a functor and make no further assumptions than that the monad laws should hold, we obtain the free monad of that functor.

Are `EvalOp` and `GPOp` functors?

```
data EvalOp b =
     DivByZeroError
   | UnknownVar String
   | VarLookup String (Int -> b)
   | VarSet String Int b
  deriving Functor
```

```
data GPOp b =
     Get (Int -> b)
   | Put Int b
  deriving Functor
```

More examples

The constant functor

Defined in `Data.Functor.Const`:

```haskell
newtype Const a b = Const {getConst :: a}

instance Functor (Const a) where
  fmap _ (Const a) = Const a
```

Question

What is `Free (Const ())`?

Question

What is `Free (Const ())`?

```
Return :: a -> Free (Const ()) a
Wrap   :: Const () (Free (Const ()) a)
             -> Free (Const ()) a
       ≈ () -> Free (Const ()) a
       ≈ Free (Const ()) a
```

Question

What is `Free (Const ())`?

```
Return :: a -> Free (Const ()) a
Wrap   :: Const () (Free (Const ()) a)
              -> Free (Const ()) a
       ≈ () -> Free (Const ()) a
       ≈ Free (Const ()) a
```

This is `Maybe`!

Question

What is `Free (Const Void)`?

Where

```
data Void
```

is an uninhabited type.

Question

What is `Free (Const Void)`?

Where

```
data Void
```

is an uninhabited type.

```
Return :: a -> Free (Const Void) a
```

But `Wrap` cannot be applied at all.

Question

What is `Free (Const Void)`?

Where

```haskell
data Void
```

is an uninhabited type.

```haskell
Return :: a -> Free (Const Void) a
```

But `Wrap` cannot be applied at all.

This is `Identity`!

Intermediate summary

- For every functor, we can obtain a monad.
- The functor describes what operations we can perform in every step.
- The values are just terms.
- Bind is substitution.

Intermediate summary

- For every functor, we can obtain a monad.
- The functor describes what operations we can perform in every step.
- The values are just terms.
- Bind is substitution.

- The primary advantage of using free monads is that we get to analyze the terms and provide possibly several interpretations of them.
- One disadvantage of free monads is that in the default definition, `(>>=)` is potentially very inefficient (can you see why)?

Several interpretations

Again, recall the exercise for `GP`:

- We had one interpretation as an `IO` action.
- We had another interpretation as a simulation, taking predefined inputs to predefined outputs.

Several interpretations

Again, recall the exercise for `GP`:

- We had one interpretation as an `IO` action.
- We had another interpretation as a simulation, taking predefined inputs to predefined outputs.

For `Eval`, we can still provide multiple interpretations:

- as a monad transformer stack,
- as a monolithic monad providing the correct functionality,
- as a combined monad using some other way to combine effects.

Another example: simulating concurrency

This idea goes back to Koen Claessen:

```
data ProcessOp :: * -> * where
  Atomically :: IO a -> (a -> r) -> ProcessOp r
  Fork       :: Process () -> r -> ProcessOp r
```

Another GADT, although just a special case: the type of the `IO` action is hidden from the result – this is often called an existentially quantified type.

Wrappers

```haskell
type Process = Free ProcessOp
```

```haskell
atomically :: IO a -> Process a
atomically m = Wrap (Atomically m Return)
fork :: Process () -> Process ()
fork p = Wrap (Fork p (Return ()))
```

Scheduling concurrent operations

```
schedule :: [Process ()] -> IO ()
schedule                              []  =
  return ()
schedule (Return _                  : ps) =
  schedule ps
schedule (Wrap (Atomically m k) : ps) =
  do
    x <- m
    schedule (ps ++ [k x])
schedule (Wrap (Fork p1 p2)       : ps) =
  schedule (ps ++ [p2, p1])
```

Example

```haskell
example :: Process ()
example = do
  fork (replicateM_ 5
    (atomically (putStrLn "Haskell")))
  fork (replicateM_ 6
    (atomically (putStrLn "cryptocurrencies")))
  atomically (putStrLn "2019")
```

Example (contd.)

```
GHCi> schedule [example]
Haskell
2019
cryptocurrencies
Haskell
cryptocurrencies
Haskell
cryptocurrencies
Haskell
cryptocurrencies
cryptocurrencies
```

Using free monads yourself

- The `Free` type (with slightly different constructor names) is provided by the `free` package.
- This also provides a `MonadFree` class and a `FreeT` monad transformer version.

Fixing the performance problem

There is a trick commonly used, corresponding to a continuation passing style transformation, which avoids the inefficient definition of `(>>=)`.

This is also implemented in the `free` package.