Finally mtl!

Joseph Tel Abrahamson / @sdbo / github.com/tel
May 23, 2015

Synopsis

- · What is "Finally Tagless"? What's important about it?
- · What is the "Monad Transformer Library", mtl?
- · mtl is a "finally tagless" effect library—and that's quite nice!

A tale of three DSLs

Hutton's Razor

```
data AddLang
  = AddLangIntLit Integer
  | Add AddLang AddLang
  deriving (Show, Eq)
interpAddLang :: AddLang → Integer
interpAddLang = \case
  AddLangIntLit i \rightarrow i
  Add l r \rightarrow interpAddLang l + interpAddLang r
addLangExp :: AddLang
addLangExp = Add (AddLangIntLit 1) (AddLangIntLit 3)
```

Hutton's Backup Razor

```
data MultLang
  = MultLangIntLit Integer
  | Mult MultLang MultLang
  deriving (Show, Eq)
interpMultLang :: MultLang → Integer
interpMultLang = \case
  MultLangIntLit i \rightarrow i
  Mult l r \rightarrow interpMultLang l * interpMultLang r
multLangExp :: MultLang
multLangExp = Mult (MultLangIntLit 1) (MultLangIntLit 3)
```

Hutton's Travel Kit

```
data RingLang
  = RingLangIntLit Integer
   | RingAdd RingLang RingLang
   | RingMult RingLang RingLang
  deriving (Show, Eq)
interpRingLang :: RingLang → Integer
interpRingLang = \case
  RingLangIntLit i \rightarrow i
  {\tt RingAdd} \quad {\tt l} \; \; r \; \rightarrow \; {\tt interpRingLang} \; \, {\tt l} \; + \; {\tt interpRingLang} \; \; r
  RingMult l r \rightarrow interpRingLang l * interpRingLang r
ringLangExp :: RingLang
ringLangExp = RingMult (RingAdd (RingLangIntLit 3)
                                       (RingLangIntLit 2))
                            (RingLangIntLit 3)
```

What's wrong with this picture?

D. R. Y.

```
data RingLang
  = RingLangIntLit Integer
  | RingAdd RingLang RingLang
  | RingMult RingLang RingLang
  deriving (Show, Eq)
interpRingLang :: RingLang \rightarrow Integer
interpRingLang = \case
  RingLangIntLit i \rightarrow i
  RingAdd l r \rightarrow interpRingLang l + interpRingLang r
  RingMult l r \rightarrow interpRingLang l * interpRingLang r
```

D. R. Y.

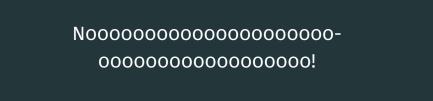
```
RingLangIntLit IntegerRingAdd RingLang RingLangRingMult RingLang RingLang
```

```
RingLangIntLit i \to i RingAdd l r \to interpRingLang l + interpRingLang r RingMult l r \to interpRingLang l * interpRingLang r
```

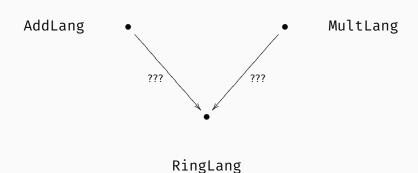
D. R. Y.

```
| RingAdd RingLang RingLang
| RingMult RingLang RingLang
```

```
RingAdd l r \rightarrow interpRingLang l + interpRingLang r RingMult l r \rightarrow interpRingLang l * interpRingLang r
```



Let's use category theory!



Once more, with feeling!

```
newtype Fix f = Fix { unFix :: f (Fix f) } fixFold :: Functor f \Rightarrow (f \ a \rightarrow a) \rightarrow (Fix \ f \rightarrow a) fixFold phi = go where go = phi . fmap go . unFix
```

```
{-# LANGUAGE DeriveFunctor #-}

data AddF x = AddI Integer | AddF x x deriving ( Show, Eq, Functor )

data MultF x = MultI Integer | MultF x x deriving ( Show, Eq, Functor )

type AddLang' = Fix AddF

type MultLang' = Fix MultF
```

```
addLangExp' :: AddLang'
addLangExp' = Fix (AddF (Fix (AddI 1)) (Fix (AddI 3)))
multLangExp' :: MultLang'
multLangExp' = Fix (MultF (Fix (MultI 1)) (Fix (MultI 3)))
```

```
-- Smart constructors! Yesssssss!
addI :: Integer → AddLang'
addI = Fix \cdot AddI
multI :: Integer → MultLang'
multI = Fix . MultI
addAdd :: AddLang′ → AddLang′ → AddLang′
addAdd l r = Fix (AddF l r)
multMult :: MultLang' → MultLang' → MultLang'
multMult l r = Fix (MultF l r)
addLangExp' :: AddLang'
addLangExp' = addAdd (addI 1) (addI 3)
multLangExp' :: MultLang'
```

1.1 .E 1 1.44 1. / 1. E 4\ / 1. E 5\

```
interpAddF :: AddF Integer → Integer
interpAddF = \case
 AddI i \rightarrow i
 AddF l r \rightarrow l + r
interpMultF :: MultF Integer → Integer
interpMultF = \case
 MultI i \rightarrow i
 MultFlr\rightarrowl*r
interpAddLang' :: AddLang' → Integer
interpAddLang' = fixFold interpAddF
interpMultLang' :: MultLang' → Integer
interpMultLang' = fixFold interpMultF
```

```
{-# LANGUAGE TypeOperators #-}
data (f :+: g) x
  = Inl(fx)
   | Inr (g x)
  deriving (Eq. Show, Functor)
-- Natural :+: eliminator
foldSum :: (f a \rightarrow a) \rightarrow (g a \rightarrow a) \rightarrow ((f :+: g) a \rightarrow a)
foldSum falg galg = \case
  Inl fa \rightarrow falg fa
  Inr ga \rightarrow galg ga
```

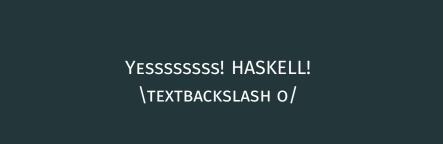
The prize!

```
type RingLang' = Fix (AddF :+: MultF)
-- More smart constructors!
ringI :: Integer → AddLang'
ringI = Fix . Inl . AddI -- why not via MultI?
ringAdd :: AddLang' → AddLang' → AddLang'
ringAdd l r = Fix (Inl (AddF l r))
ringMult :: MultLang' → MultLang' → MultLang'
ringMult l r = Fix (Inr (MultF l r))
ringLangExp' :: RingLang'
ringLangExp' = ringMult (ringAdd (ringI 3) (ringI 2)) (ringI 3)
interpRingLang' :: RingLang' → Integer
interpRingLang' = fixFold (foldSum interpAddF interpMultF)
```

20/65

Prelude> interpRingLang' ringLangExp'

Prelude> interpRingLang' ringLangExp'
15



D. R. Y. Stats

40 additions, 3 new PRAGMAs

D. R. Y. Stats

40 additions, 3 new PRAGMAs

Used **Fix** in anger

YESSSSSSS! HASKELL! (+1 Functional Programming)

In all seriousness...

Generalizing :+:

Wouter Swierstra, Data types à la carte.

Really cool.

and Shan

Data types à la Carette, Kiselyov,

Let's try this again...

Design by wishful thinking

```
class Adds v where  \text{add} :: v \to v \to v  class Multiplies v where  \text{mult} :: v \to v \to v
```

```
instance Adds Integer where add = (+)
instance Multiplies Integer where mult = (*)
addsExp :: Integer
addsExp = add 1 3
multsExp :: Integer
multsExp = mult 1 3
```

class FromInteger v where

 $\mathtt{i}\, ::\, \mathbf{Integer}\, \rightarrow\, \mathtt{v}$

instance FromInteger Integer where

i = id

```
addsExp :: (FromInteger v, Adds v) \Rightarrow v addsExp = add (i 1) (i 3) multsExp :: (FromInteger v, Multiplies v) \Rightarrow v multsExp = mult (i 1) (i 3)
```

```
{-# LANGUAGE ConstraintKinds #-} 
type Rings v = (FromInteger v, Adds v, Multiplies v) 
ringsExp :: Rings v \Rightarrow v 
ringsExp = mult (add (i 3) (i 2)) (i 3)
```

Prelude> ringsExp

Prelude> ringsExp 15 Prelude> ringsExp :: Integer
15

But then!

```
instance FromInteger RingLang where i = RingLangIntLit
instance Adds RingLang where add = RingAdd
instance Multiplies RingLang where mult = RingMult
```

Prelude> ringsExp :: RingLang

· One-line composabilty

- · One-line composabilty
- · Needn't mention concrete data types or Inl/Inr

- · One-line composabilty
- · Needn't mention concrete data types or Inl/Inr
 - · "tagless"

- · One-line composabilty
- · Needn't mention concrete data types or Inl/Inr
 - · "tagless"
- · But if we have them, we can still recover "raw" ASTs

"Does is scale?"

Yes.

Yes. Well, but don't trust me.

The "Monad Transformer Library"

Or: "Haskell2010 Composable Effects Through Prolog Technology"

Or: "Haskell2010 Composable Effects Through Typeclass Technology"

import Control.Monad.State

```
inc :: MonadState Int m ⇒ m Int
inc = do
  count ← get
  set (count + 1)
  return count
```

-- let's get a concrete stack from 'transformers'
import Control.Monad.Trans.State (runState)

Prelude> flip runState 0 inc
(1, 1)

-- let's get a concrete stack from 'transformers'
import Control.Monad.Trans.State (runState)

Prelude> flip runState 0 (inc :: StateT Int Identity Int)
(1, 1)

Solve your monad stacks at compile time!

```
op :: ( MonadState S m, MonadError E m, MonadIO m ) \Rightarrow m A op :: ( MonadIO m ) \Rightarrow FreeT (StateErrorF S E) m A op :: FreeT (StateErrorF S E) IO A op :: StateT S (EitherT E IO) A op :: EitherT E (StateT S IO) A
```

Complaints

Just a few points in the lattice

```
Monad m \Rightarrow Monad (StateT s m)
Functor m \Rightarrow Functor (StateT s m)
MonadTrans (StateT s)
```

MonadCont (ContT r m) Source

```
MonadCont (ContT r m) Source
MonadCont m \Rightarrow MonadCont (MaybeT m) Source
MonadCont m ⇒ MonadCont (ListT m) Source
MonadCont m ⇒ MonadCont (IdentityT m) Source
(Monoid w, MonadCont m) ⇒ MonadCont (WriterT w m) Source
(Monoid w, MonadCont m) ⇒ MonadCont (WriterT w m) Source
(Error e, MonadCont m) \Rightarrow MonadCont (ErrorT e m) Source
MonadCont m \Rightarrow MonadCont (ExceptT e m) Source
MonadCont m \Rightarrow MonadCont (StateT s m) Source
MonadCont m \Rightarrow MonadCont (StateT s m) Source
MonadCont m \Rightarrow MonadCont (ReaderT r m) Source
(Monoid w, MonadCont m) \Rightarrow MonadCont (RWST r w s m) Source
(Monoid w. MonadCont m) \Rightarrow MonadCont (RWST r w s m) Source
-- This gets quite complex
```

N classes and M concrete types to NM instance definitions

N classes and M concrete types to NM instance definitions $O(n^2)$

Right?

Right?

· Well, not for library authors anyway

Right?

- · Well, not for library authors anyway
- This technique is thus packaging resistent—harder to learn about

Right?

- · Well, not for library authors anyway
- This technique is thus packaging resistent—harder to learn about
- · But for users writing their own classes, N and M remain small

Teaching your own pet Prolog

· Instances are typeclass Prolog *predicates*

Teaching your own pet Prolog

- · Instances are typeclass Prolog predicates
- · Effects don't just commute: the theory is hard to generalize today

Teaching your own pet Prolog

- · Instances are typeclass Prolog predicates
- Effects don't just commute: the theory is hard to generalize today
- · mtl lets/forces you to write down all of what you know

Lose control of ordering

```
op :: (MonadError e m, MonadState s m) \Rightarrow m ()
```

Lose control of ordering

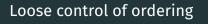
```
op :: (MonadError e m, MonadState s m) \Rightarrow m () 
op =\Rightarrow StateT s (Either e) () 
op =\Rightarrow EitherT e (State s) ()
```

Lose control of ordering

```
op :: (MonadError e m, MonadState s m) \Rightarrow m () op =\Rightarrow StateT s (Either e) () op =\Rightarrow EitherT e (State s) () No right answer!
```

Loose control of ordering

This is why we have laws.



This is why we have laws. Combining laws "correctly" is hard.

Loose control of ordering

 $\textbf{class} \text{ (MonadState s m, MonadError e m)} \Rightarrow \textbf{MonadParser s e m where } \{\}$

Tight denotation of semantics

```
-- Tighten to your domain

class MonadParser m where

type family Char m :: *

type family Error m :: *

peekChar :: m (Char m)

getChar :: m (Char m)

failParse :: Error m → m a
```

One-step reduction only

```
| Neg Negated  
 optimize :: Negated \rightarrow Negated optimize (Neg (Neg n)) = optimize n optimize n = n
```

One-step reduction only

```
class Negated v where
  int :: Integer → v
  neg :: v → v

instance Negated Integer where
  int = id
  neg i = - i
```

One-step reduction only

```
data Neg = Neg { negating :: Bool, value :: Integer }
instance Negated Neg where
  int i = Neg { negating = False, value = i }
  neg n = n { negating = not (negating n) }

runNeg :: Neg → Integer
runNeg n = if negating n then - (value n) else value n
```

Conclusions

 When explicit structure's got you down, specifying your needs in Prolog might be worth a shot

- When explicit structure's got you down, specifying your needs in Prolog might be worth a shot
- · Finally Tagless style can represent whatever AST fragments you like

- When explicit structure's got you down, specifying your needs in Prolog might be worth a shot
- · Finally Tagless style can represent whatever AST fragments you like
- mtl is a mature example of using Finally Tagless at scale in a library

- When explicit structure's got you down, specifying your needs in Prolog might be worth a shot
- · Finally Tagless style can represent whatever AST fragments you like
- mtl is a mature example of using Finally Tagless at scale in a library
 - · As a user, it gives you what you want

- When explicit structure's got you down, specifying your needs in Prolog might be worth a shot
- · Finally Tagless style can represent whatever AST fragments you like
- mtl is a mature example of using Finally Tagless at scale in a library
 - · As a user, it gives you what you want
 - · Maintainers bear a big burden

- When explicit structure's got you down, specifying your needs in Prolog might be worth a shot
- · Finally Tagless style can represent whatever AST fragments you like
- mtl is a mature example of using Finally Tagless at scale in a library
 - · As a user, it gives you what you want
 - · Maintainers bear a big burden
 - Good demonstration of why composition is hard

- When explicit structure's got you down, specifying your needs in Prolog might be worth a shot
- · Finally Tagless style can represent whatever AST fragments you like
- mtl is a mature example of using Finally Tagless at scale in a library
 - · As a user, it gives you what you want
 - · Maintainers bear a big burden
 - · Good demonstration of why composition is hard
- Try them out!

Thanks!

Tweet at me!

@sdbo

Extra Slides

class Ast m where

```
\begin{array}{lll} k & :: a \rightarrow m \ a \\ \\ lam :: (m \ a \rightarrow m \ b) \rightarrow m \ (a \rightarrow b) \\ \\ app :: m \ (a \rightarrow b) \rightarrow (m \ a \rightarrow m \ b) \end{array}
```

instance Ast Ast where ...

```
newtype Eval a = Eval { eval :: a }
instance Ast Eval where
  k = Eval
  lam f = Eval (eval . f . Eval)
  app (Eval f) = Eval . f . eval
```

```
class Adder m where
   \mathsf{adder}\, ::\, \mathsf{m}\, \, \mathsf{Int}\, \to \, \mathsf{m}\, \, \mathsf{Int}\, \to \, \mathsf{m}\, \, \mathsf{Int}\,
instance Adder Eval where
   adder (Eval a) (Eval b) = Eval (a + b)
ex :: (Adder m, Ast m) \Rightarrow m Int
ex = app (lam (adder (k 3))) (k 4)
-- Prelude> eval ex
-- 7
```