Finally `mtl`!

Joseph Tel Abrahamson / @sdbo / `github.com/tel`

May 22, 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

```haskell
data AddLang
  = AddLangIntLit Integer
  | Add AddLang AddLang
  deriving ( Show, Eq )

interpAddLang :: AddLang → Integer
interpAddLang = \case
  AddLangIntLit i → i
  Add l r  → interpAddLang l + interpAddLang r

addLangExp :: AddLang
addLangExp = Add (AddLangIntLit 1) (AddLangIntLit 3)
```

Hutton's Backup Razor

```haskell
data MultLang
  = MultLangIntLit Integer
  | Mult MultLang MultLang
  deriving ( Show, Eq )

interpMultLang :: MultLang → Integer
interpMultLang = \case
  MultLangIntLit i → i
  Mult l r → interpMultLang l ∗ interpMultLang r

multLangExp :: MultLang
multLangExp = Mult (MultLangIntLit 1) (MultLangIntLit 3)
```

Hutton's Travel Kit

```haskell
data RingLang
  = RingLangIntLit Integer
  | RingAdd  RingLang RingLang
  | RingMult RingLang RingLang
  deriving ( Show, Eq )

interpRingLang :: RingLang → Integer
interpRingLang = \case
  RingLangIntLit i → i
  RingAdd  l r → interpRingLang l + interpRingLang r
  RingMult l r → interpRingLang l ∗ interpRingLang r

ringLangExp :: RingLang
ringLangExp = RingMult (RingAdd (RingLangIntLit 3)
                                (RingLangIntLit 2))
                       (RingLangIntLit 3)
```

What’s wrong with this picture?

```haskell
data RingLang
  = RingLangIntLit Integer
  | RingAdd  RingLang RingLang
  | RingMult RingLang RingLang
  deriving ( Show, Eq )

interpRingLang :: RingLang → Integer
interpRingLang = \case
  RingLangIntLit i → i
  RingAdd  l r → interpRingLang l + interpRingLang r
  RingMult l r → interpRingLang l ∗ interpRingLang r
```

```
= RingLangIntLit Integer
| RingAdd  RingLang RingLang
| RingMult RingLang RingLang
```

```
RingLangIntLit i → i
RingAdd  l r → interpRingLang l + interpRingLang r
RingMult l r → interpRingLang l * interpRingLang r
```

D. R. Y.

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

```
RingAdd  l r → interpRingLang l + interpRingLang r
RingMult l r → interpRingLang l * interpRingLang r
```

Nooooooooooooooooooooo-
oooooooooooooooooo!

Let's use category theory!

$$
\begin{tikzcd}
\texttt{AddLang} \arrow[dr, "???"'] & & \texttt{MultLang} \arrow[dl, "???"] \\
& \texttt{RingLang} &
\end{tikzcd}
$$

Once more, with feeling!

Routine recursion scheme surgery

```haskell
newtype Fix f = Fix { unFix :: f (Fix f) }

fixFold :: Functor f ⇒ (f a → a) → (Fix f → a)
fixFold phi = go where go = phi . fmap go . unFix
```

Routine recursion scheme surgery

```haskell
{-# LANGUAGE UndecidableInstances #-}

deriving instance Show (f (Fix f)) => Show (Fix f)
deriving instance Eq   (f (Fix f)) => Eq   (Fix f)
```

Routine recursion scheme surgery

```
{-# 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)))
```

Routine recursion scheme surgery

```
-- Smart constructors! Yesssssss!
addI :: Integer → AddLang'
addI = Fix . 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'
```

```
interpAddF :: AddF Integer → Integer
interpAddF = \case
  AddI i   → i
  AddF l r → l + r

interpMultF :: MultF Integer → Integer
interpMultF = \case
  MultI i   → i
  MultF l r → l ∗ r

interpAddLang' :: AddLang' → Integer
interpAddLang' = fixFold interpAddF

interpMultLang' :: MultLang' → Integer
interpMultLang' = fixFold interpMultF
```

```haskell
{-# LANGUAGE TypeOperators #-}

data (f :+: g) x
  = Inl (f x)
  | Inr (g x)
  deriving ( Eq, Show, Functor )

-- Natural :+: eliminator
foldSum :: (f a → a) → (g a → a) → ((f :+: g) a → a)
foldSum falg galg = \case
  Inl fa → falg fa
  Inr ga → 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)
```

```
Prelude> interpRingLang' ringLangExp'
```

```
Prelude> interpRingLang' ringLangExp'
15
```

Yesssssssss! HASKELL!
\textbackslash o/

D. R. Y. Stats

40 additions, 3 new PRAGMAs

40 additions, 3 new PRAGMAs

Used `Fix` in anger

Yesssssssss! HASKELL!
(+1 Functional Programming)

In all seriousness…

Generalizing `:+:`

Wouter Swierstra, *Data types à la carte*.

Really cool.

Data types à la Carette, Kiselyov, and Shan

Let’s try this again…

Design by wishful thinking

```
class Adds v where
  add :: v → v → v

class Multiplies v where
  mult :: v → v → v
```

```haskell
instance Adds Integer where add = (+)
instance Multiplies Integer where mult = (*)

addsExp :: Integer
addsExp = add 1 3

multsExp :: Integer
multsExp = mult 1 3
```

```haskell
class FromInteger v where
  i :: Integer → v

instance FromInteger Integer where
  i = id
```

```
addsExp :: (FromInteger v, Adds v) ⇒ v
addsExp = add (i 1) (i 3)

multsExp :: (FromInteger v, Multiplies v) ⇒ v
multsExp = mult (i 1) (i 3)
```

```haskell
{-# LANGUAGE ConstraintKinds #-}

type Rings v = (FromInteger v, Adds v, Multiplies v)

ringsExp :: Rings v ⇒ 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
```

```
Prelude> ringsExp :: RingLang
RingMult (RingAdd (RingLangIntLit 3)
                  (RingLangIntLit 2))
         (RingLangIntLit 3)
```

Recap

- One-line composabiity

Recap

- One-line composabiity
- Needn't mention concrete data types or `Inl`/`Inr`

Recap

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

Recap

- One-line composabiity
- 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.

The ”Monad Transformer Library”

Or: “Haskell2010 Composable Effects Through Prolog Technology”

Or: “Haskell2010 Composable Effects Through Typeclass Technology”

```haskell
import Control.Monad.State

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

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

Prelude> flip runState 0 inc
(1, 1)
```

```haskell
-- 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!

Solve your monad stacks at compile time! Not runtime

Solve your monad stacks at compile time! Not runtime, nor write-time.

Complaints

Just a few points in the lattice

```
MonadTrans (ContT r) Source
Monad (ContT r m) Source
Functor (ContT r m) Source
Applicative (ContT r m) Source
MonadIO m ⇒ MonadIO (ContT r m) Source
```

All points in the lattice

`MonadCont (ContT r m) Source`

All points in the lattice

```
MonadCont (ContT r m) Source

MonadCont m ⇒ 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) ⇒ MonadCont (ErrorT e m) Source
MonadCont m ⇒ MonadCont (ExceptT e m) Source
MonadCont m ⇒ MonadCont (StateT s m) Source
MonadCont m ⇒ MonadCont (StateT s m) Source
MonadCont m ⇒ MonadCont (ReaderT r m) Source
(Monoid w, MonadCont m) ⇒ MonadCont (RWST r w s m) Source
(Monoid w, MonadCont m) ⇒ MonadCont (RWST r w s m) Source
...
-- This gets quite complex
```

All points in the lattice

But this is just the name of the game! Effects don’t commute.

Lose control of ordering

```
op :: (MonadError e m, MonadState s m) ⇒ m ()
```

Lose control of ordering

```
op :: (MonadError e m, MonadState s m) => m ()

op ==> StateT s (Either e) ()
op ==> EitherT e (State s) ()
```

Lose control of ordering

```
op :: (MonadError e m, MonadState s m) ⇒ m ()

op =⇒ StateT s (Either e) ()
op =⇒ 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

```
class (MonadState s m, MonadError e m) ⇒ 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
```

Thanks!

Tweet at me!

@sdbo