

Finally mt l!

Joseph Tel Abrahamson

May 20, 2015

- What is “Finally Tagless”? What’s important about it?
- What is the “Monad Transformer Library”, `mtl`?
- `mtl` is a “finally tagless” effect library
- Comparing effect handlers

A tale of three DSLs

```
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)
```

```
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)
```

```
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?

```
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
```

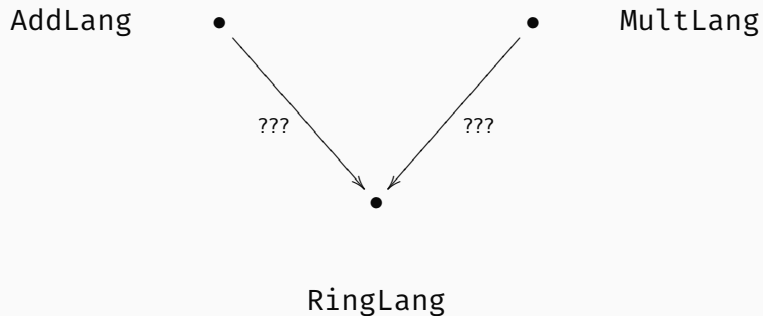
```
RingLangIntLit i → i  
RingAdd  l r → interpRingLang l + interpRingLang r  
RingMult l r → interpRingLang l * interpRingLang r
```

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

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

Noooooooooooooooooooooo-
oooooooooooooooooooo!

Let's use category theory!



Once more, with feeling!

```
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
```

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

```
deriving instance Show (f (Fix f))  $\Rightarrow$  Show (Fix f)
```

```
deriving instance Eq (f (Fix f))  $\Rightarrow$  Eq (Fix f)
```

```
{-# 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! Yessssssss!
```

```
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'
```

```
multLangExp' = multMult (multI 1) (multI 3)
```

Routine recursion scheme surgery

`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`

```
{-# 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 MultiI?

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
```

YESSSSSSSS! HASKELL!
\\TEXTBACKSLASH O/

40 additions, 3 new PRAGMAs

40 additions, 3 new PRAGMAs
Used **Fix** in anger

YESSSSSSSS! HASKELL!
(+1 FUNCTIONAL PROGRAMMING)

Generalizing $:+:$

Wouter Swierstra, *Data types à la carte*.

Really cool.

Data types à la Carette, Kiselyov, and Shan

Let's try this again...

```
class Adds v where
```

```
  add :: v → v → v
```

```
class Multiplies v where
```

```
  mult :: v → v → 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
```

```
  i :: Integer → 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
```

```
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)
```


- One-line composability

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

- One-line composability
- Needn't mention concrete data types or `Inl/Inr`
 - “tagless”

- One-line composability
- 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"
