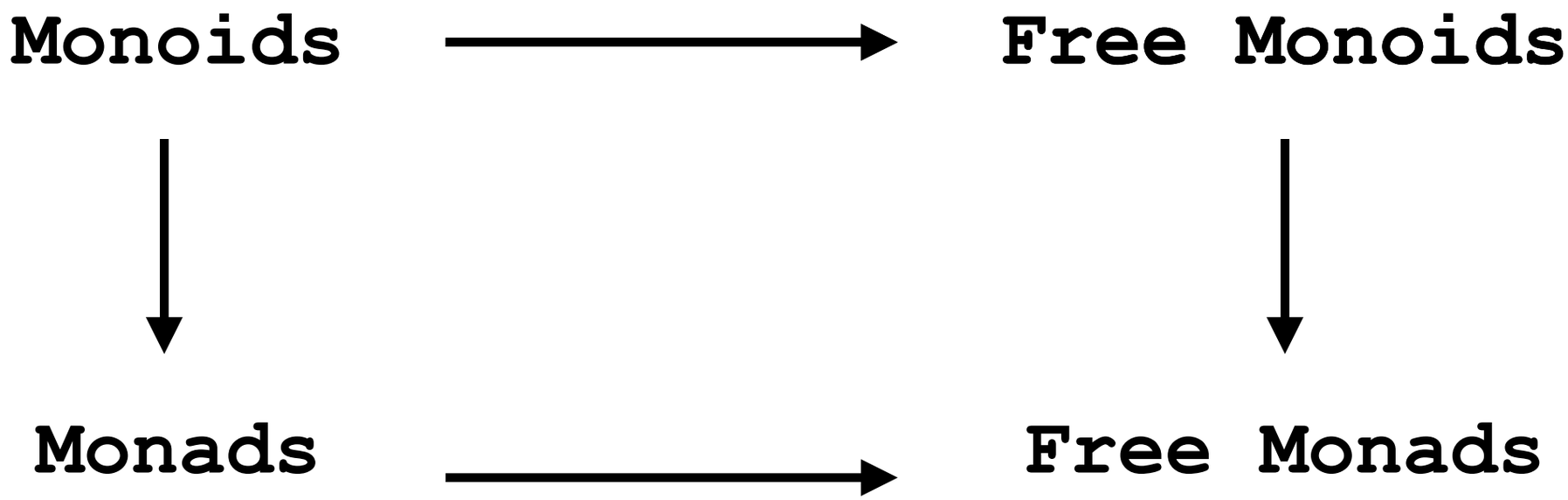
Free, but not as in beer or speech

Dan Piponi

The Plan



Magma

Magmas

```
class Magma m where
  o :: m -> m -> m
```

```
instance Magma Integer where
  o = (+)
```

Testing magmas

```
accumulate :: Magma m => m -> [m] -> m
accumulate a [] = a
accumulate a (b : bs) = accumulate (a `o` b) bs
```

```
accumulate 0 [1, 2, 3] == 6
accumulate 10 [-10, 5] == 5
```

Testing magmas

```
accumulate' :: Magma m => m -> [m] -> m
accumulate' a [] = a
accumulate' a (b : bs) = accumulate' (b `o` a) bs
```

```
accumulate' 0 [1, 2, 3] == 6
accumulate' 10 [-10, 5] == 5
```

```
a `o` b == b `o` a
```

The least special magma

Suppose $a,b \in M$ and that $a \neq b$

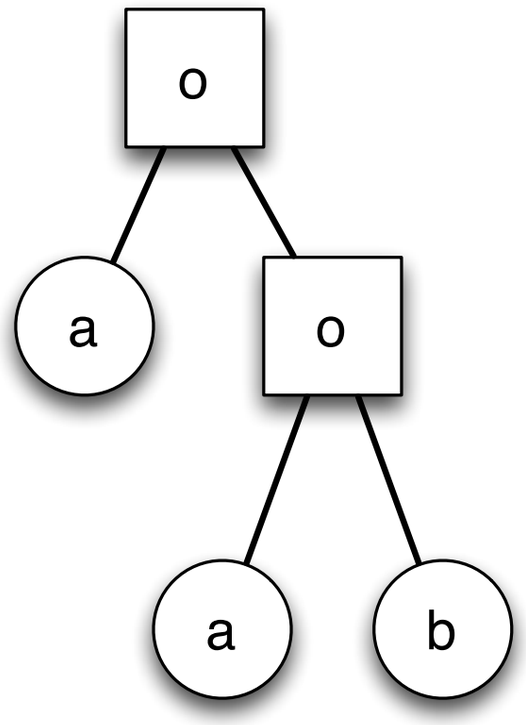
We know $ab \in M$.
We'd like $ab \neq a$ and $ab \neq b$

We also know $a(ab) \in M$.
We'd like $a(ab) \neq a$ and $a(ab) \neq b$ and $a(ab) \neq ab$

And so on…

The free magma

a(ab)




```
data FreeMagma a = Var a
                 | Tree (FreeMagma a) (FreeMagma a)

instance Magma (FreeMagma a) where
   o = Tree
```

The essence of freeness

```
interpretMagma :: Magma b => (a -> b) ->
                  (FreeMagma a -> b)

interpretMagma f (Var a) = f a
interpretMagma f (Tree a b) =
  interpretMagma f a `o`
  interpretMagma f b
```

An element of FreeMagma a is an “expression” in a bunch of variables using the language of magmas. The interpret function evaluates these expressions given a rule saying how the elements of a are to be replaced by values in a “real” magma.

Freeness at work

Evaluate a(ab) in Integer
given a = 1
b = 2

```
x :: FreeMagma String
x = Tree (Var "a") (Tree (Var "a") (Var "b"))

f :: String -> Integer
f "a" = 1
f "b" = 2

test0 = interpMagma f x
```

Monoids

```
class Monoid m where
   (<>) :: m -> m -> m
   mempty :: m
```

```
instance Monoid Integer where
   (<>) = (+)
   mempty = 0
```

Monoid laws

```
a <> (b <> c) == (a <> b) <> c
```

```
a <> mempty == mempty <> a == a
```

Free Monoid

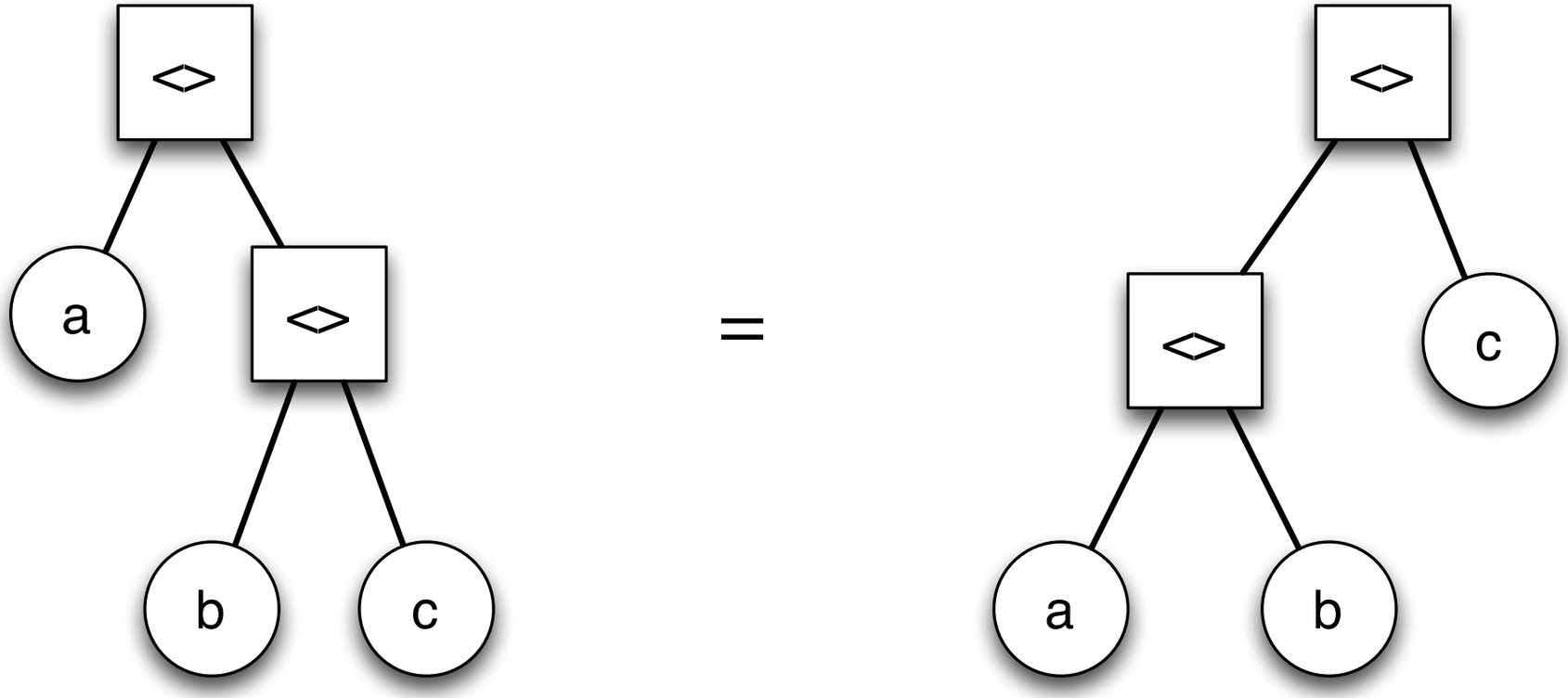
```
data FreeMonoid a = Var a
                  | MEmpty
                  | Tree (FreeMonoid a) (FreeMonoid a)
```

```
instance Monoid (FreeMonoid a) where
   (<>) = Tree
   mempty = MEmpty
```

Fails to satisfy monoid laws. Eg. **`Tree MEmpty a /= a`**

Using the laws

a(bc) = (ab) c



Using the laws

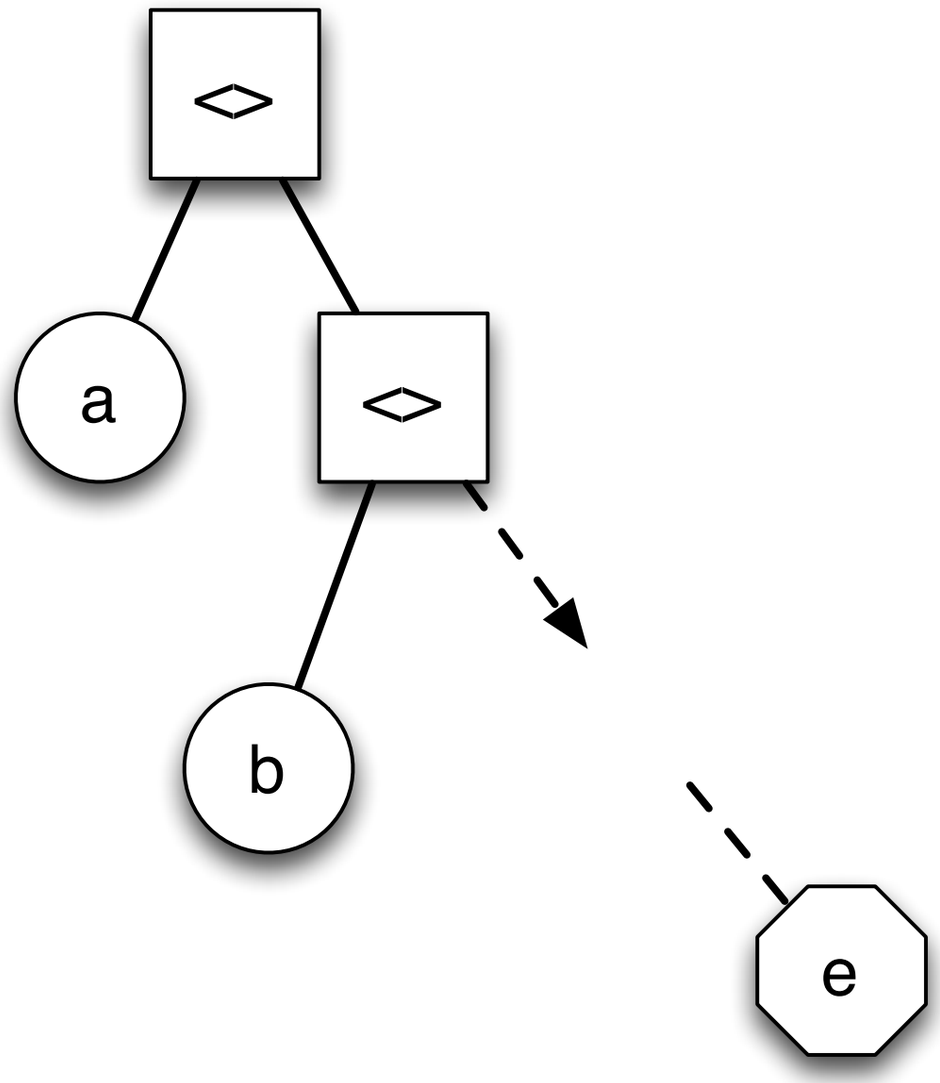
$$((ab)c)d$$

$$= (ab)(cd)$$

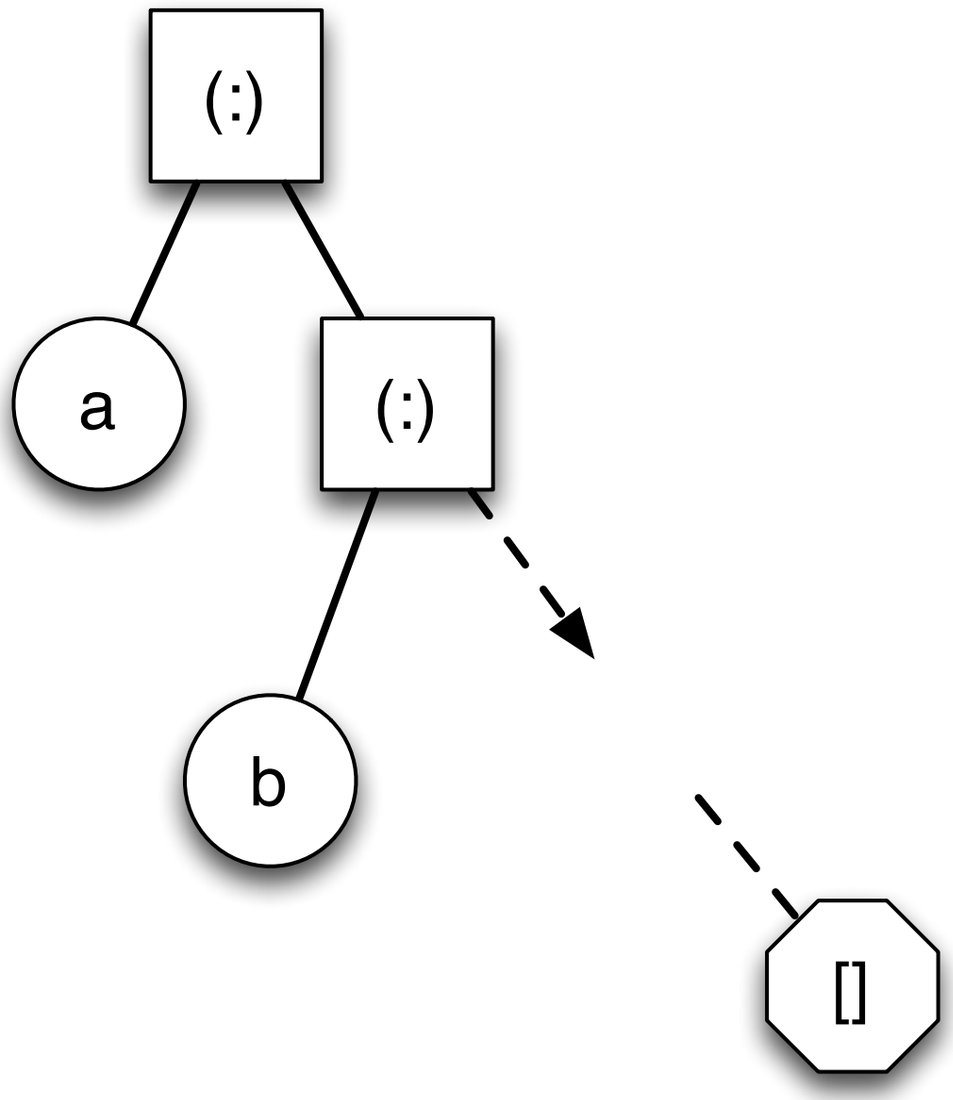
$$= a(b(cd))$$

$$= a(b(c(de)))$$

Using the laws



Lists



Lists are free monoids

```
instance Monoid [a] where
   (*) = (++)
   e = []
```

Satisfy all the monoid laws

More essence of freeness

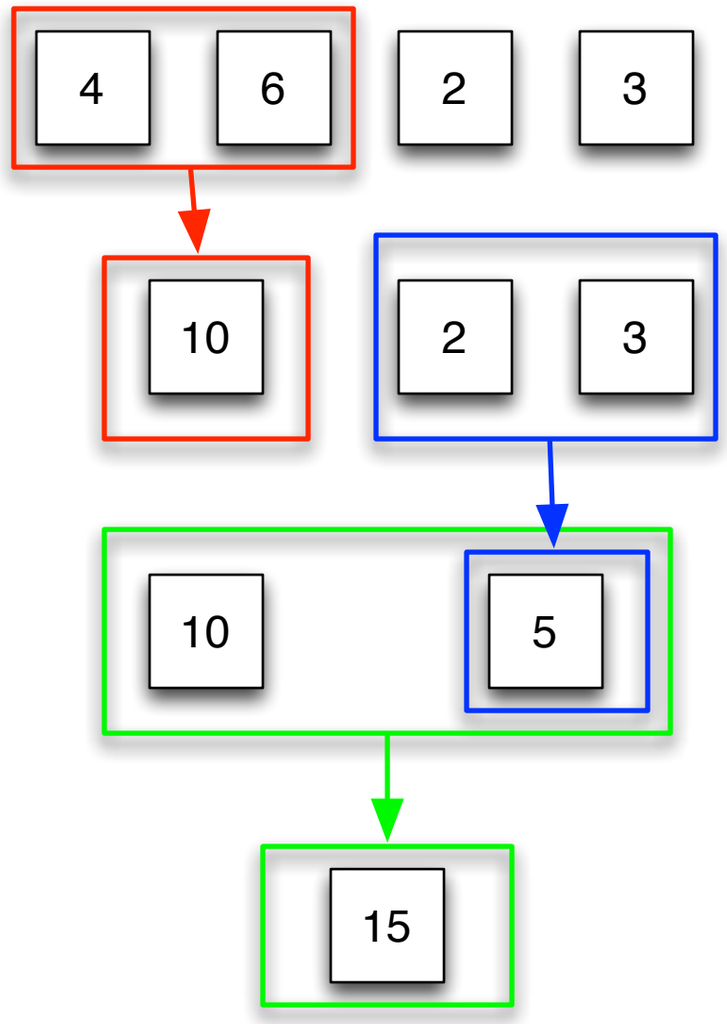
```
InterpretMonoid :: Monoid b => (a -> b) -> ([a] -> b)
```

```
interpretMonoid f [] = e
InterpretMonoid f (a : as) = f a <> interpretMonoid f as
```

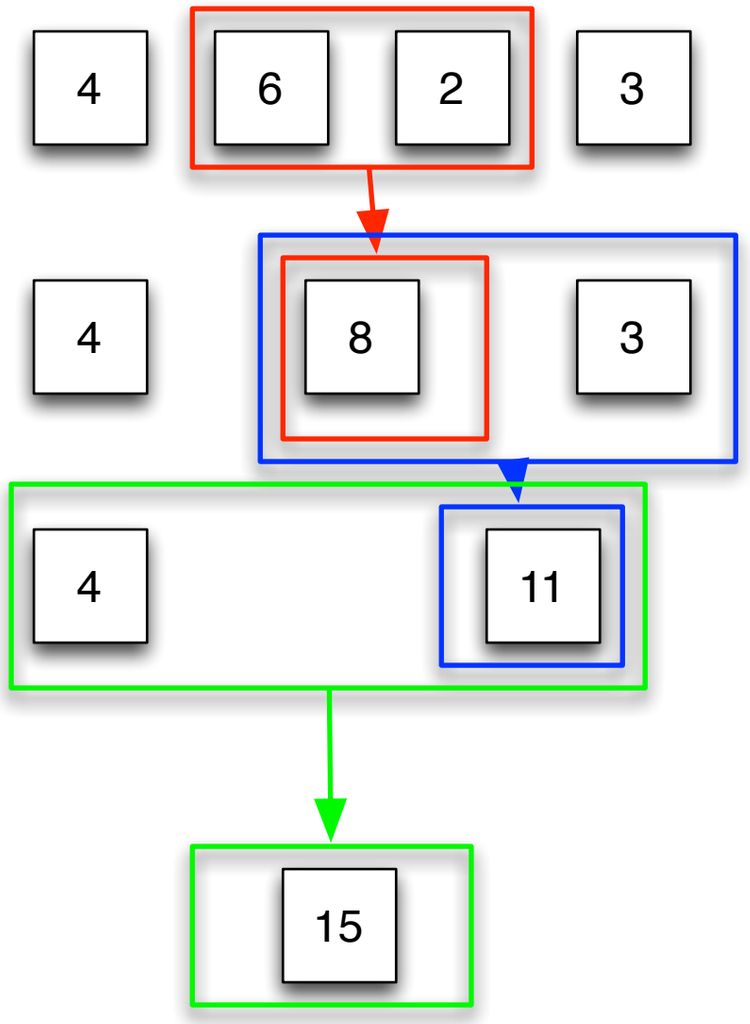
Syntax and semantics

```
data BankOp = Deposit Integer | Widthdraw Integer

program =
   [Deposit 10, Widthdraw 5, Withdraw 2, Deposit 3]

interpretOp :: BankOp -> Integer
interpretOp (Deposit d) = d
interpretOp (Withdraw w) = -w

interpret = interpretMonoid interpretOp


interpret program = 6
```

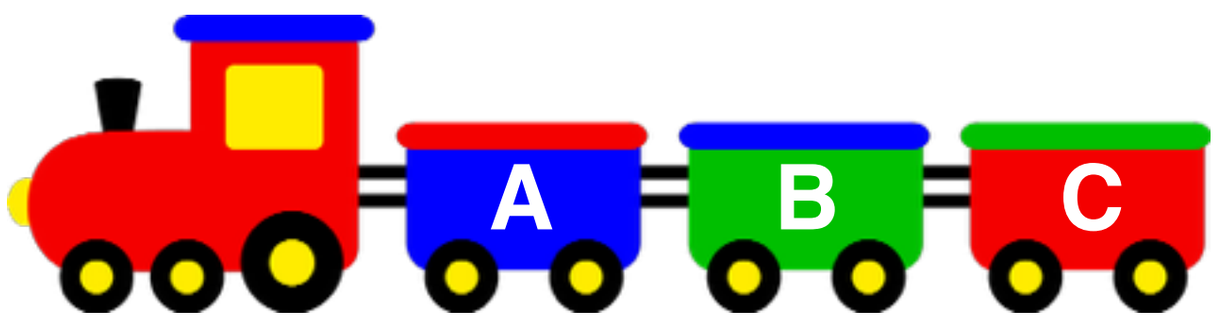
Associativity



or

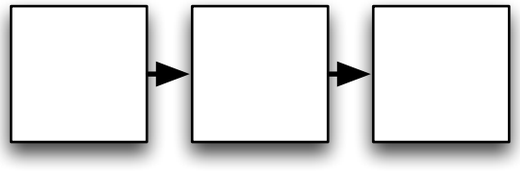


Monoids are limited

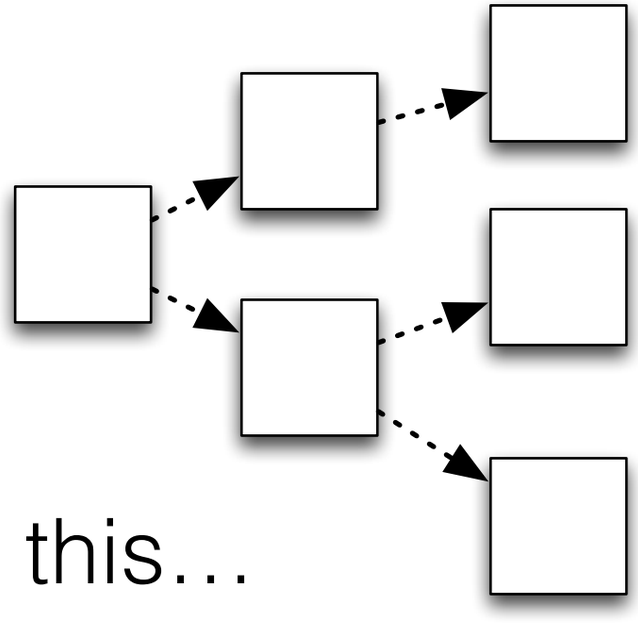


- Limited to a linear sequence of operations
- What if we want to take one path or another based on some kind of input?

Instead of this…



…we want this…



How to design such a thing?

- We want to generalise the product a<>b to something like a<>container of b's
- Tricky because we might want different a's to accept different number of b's on the right
- A solution is to make the elements of our new structure themselves containers
- Our multiplication operation will no longer be a binary operation - it will be unary because its argument contains both the a and the b's, as the a contains the b's

How to design such a thing?

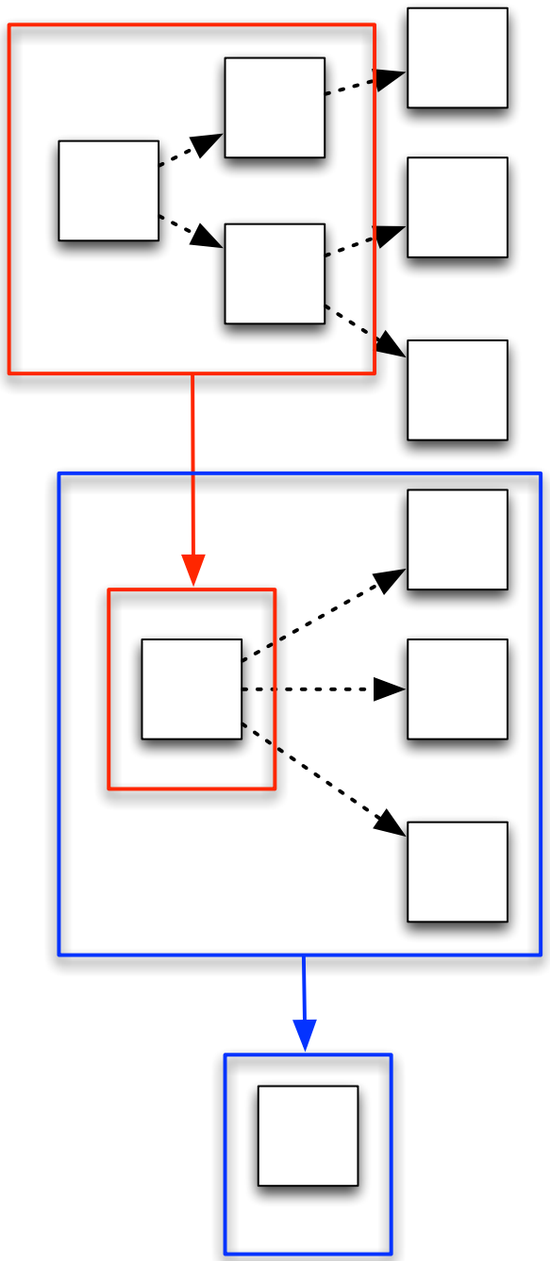
- We'd like an identity element.
- To be similar to the monoid identity we'd like to be able to multiply by a single element on the right.
- So the identity element must be a container, albeit with one element inside it.

Let’s sketch the interface

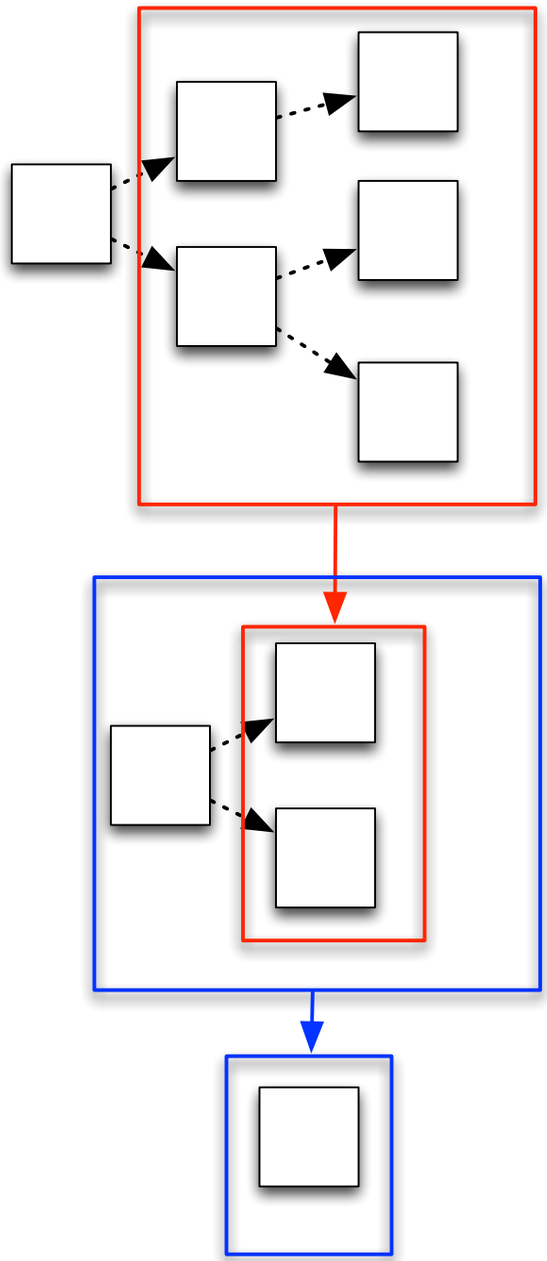
```
class Functor m => MonoidyThing m where
   identity :: a -> m a
   multiply :: m (m a) -> m a
```

```
class Functor m => Monad m where
   return :: a -> m a
   join :: m (m a) -> m a


join x = x >>= id
x >>= f = join (fmap f x)
```

What about some laws?



`join . join`



`join . fmap join`

So what's a free monad?

- Just as multiplication in the free monoid builds lists, multiplication should build trees.
- Just as free monoids are built from some base type of elements, free monads should be built from some base container type, along with an identity container.

Implementing the free monad

- If X = [a], then X is a solution to the equation
  $X = 1+a\times X$
- Implemented as
  `data List a = Nil | Cons a (List a)`
- We want a solution to the equation
  $X = e+f\ X$
- Remembering that the free monad itself must be a container type, implemented as
  `data Free f a = Id a | Free (f (Free f a))`

Implement join?

```
data Free f a = Id a | Free (f (Free f a))

join' :: Functor f =>
         Free f (Free f a) -> Free f a
join' (Id x) = x
join' (Free fa) = Free (fmap join' fa)
```

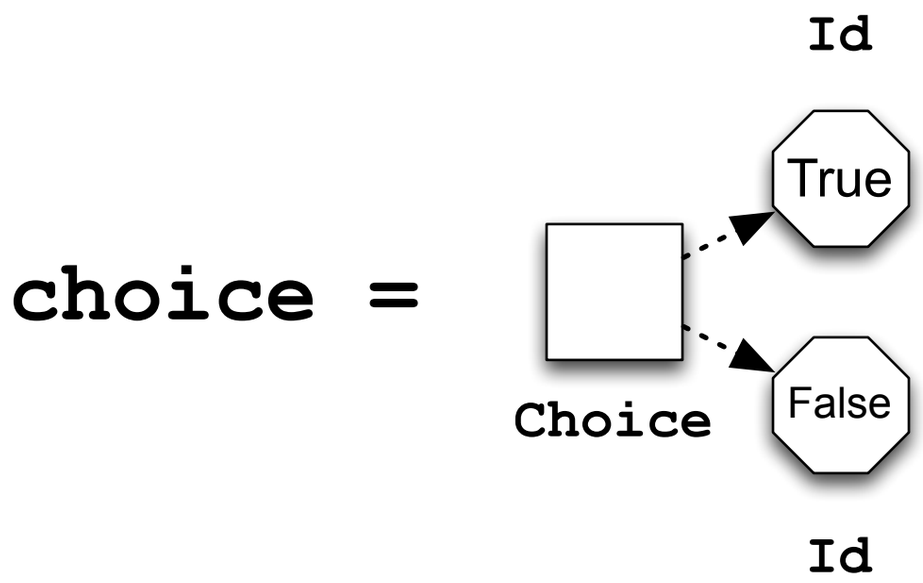
What does **`interpretMonad`** look like?

```
instance Functor f => Functor (Free f) where
    fmap f (Id x)     = Id (f x)
    fmap f (Free fa) = Free (fmap (fmap f) fa)
```

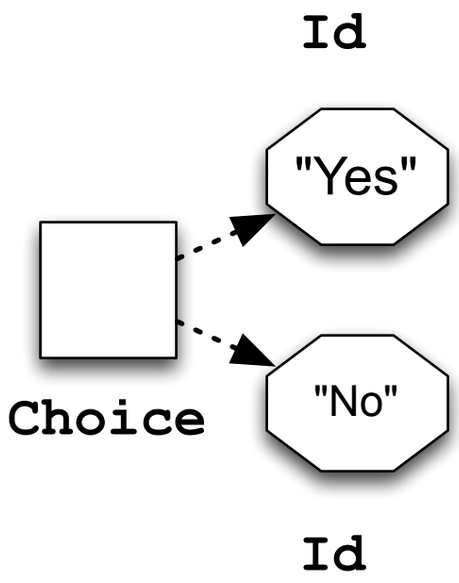
```
interpretMonad :: (Functor a, Functor b, Monad b) =>
                  (forall x.a x -> b x) ->
                  (forall x.Free a x -> b x)

interpretMonad f (Id x)     = return x
interpretMonad f (Free ax) =
        join (f (fmap (interpretMonad f) ax))
```

An example: binary choice

```
data Choice x = Choice x x

instance Functor Choice where
    fmap f (Choice x y) = Choice (f x) (f y)

choice :: Free Choice Bool
choice = Free (Choice (Id True) (Id False))
```

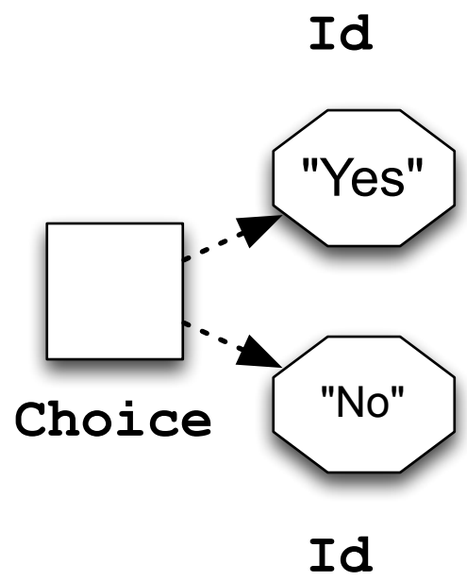


Choosing the leaves




```
fmap (\b -> if b then “Yes” else “No”)
     choice
```

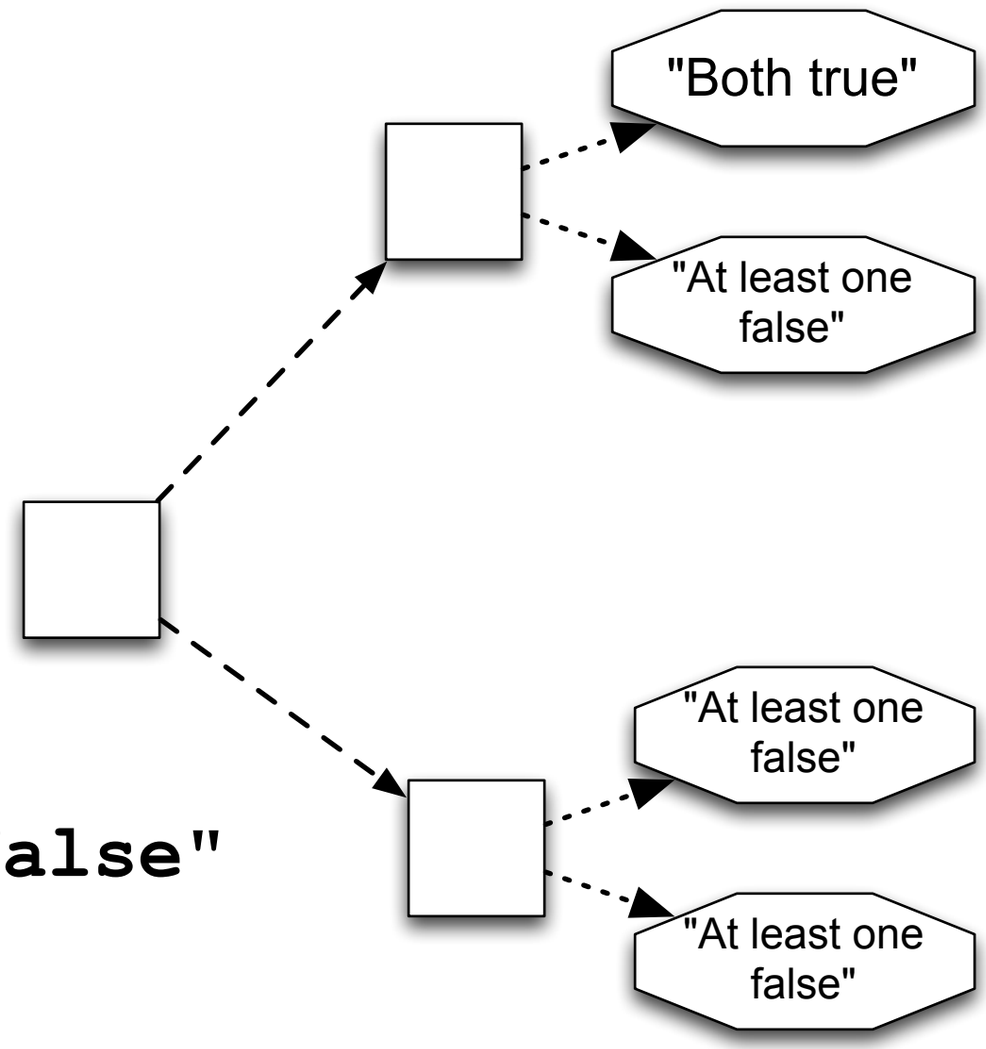
Choosing the leaves



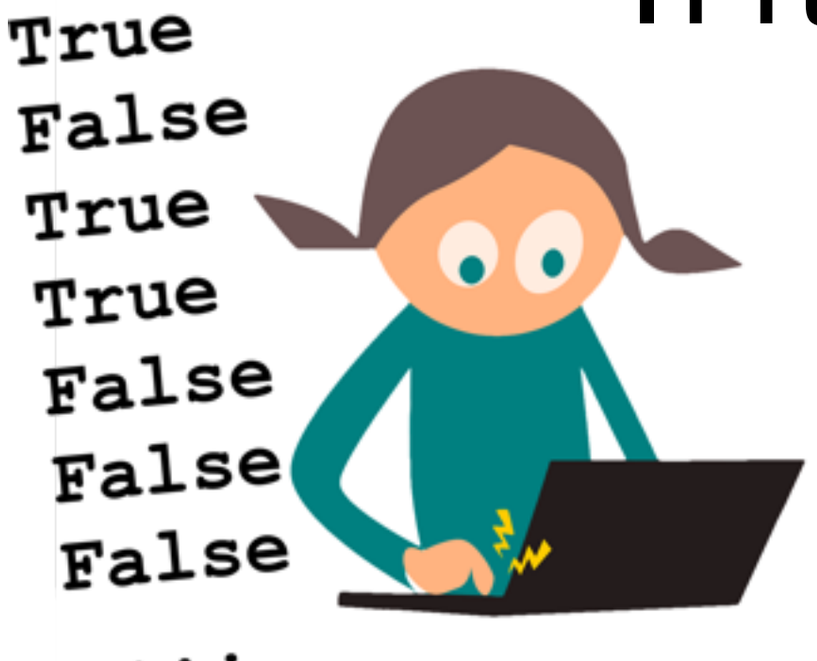

```
do
   x <- choice
   return $ if x then “Yes” else “No”
```

Using **Free Choice**

```
test :: Free Choice String
test = do
     a <- choice
     b <- choice
     return $ if a && b
         then "Both true”
         else "At least one false"
```



Interpreter #1




```
inter1 :: Choice x -> IO x
inter1 (Choice a b) = do
    c <- readLn :: IO Bool
    return $ if c then a else b

go1 = interpretMonad inter1 test
```

Interpreter #2

`True`

```
inter2 :: Choice x -> Reader Bool x
inter2 (Choice a b) = do
     c <- ask
     return $ if c then a else b

go2 = runReader
   (interpretMonad inter2 test) True
```

Interpreter #3

```
[True, False, True, True, False, ...]
```

```
inter3 :: Choice x -> State [Bool] x
inter3 (Choice a b) = do
    x : xs <- get
    put xs
    return $ if x then a else b

go3 = evalState (interpretMonad inter3 test)
                [True, False]
```

Interpreter #3’

```
inter3' :: Free Choice x -> State [Bool] x
inter3' (Id a) = return a
inter3' (Free (Choice a b)) = do
    x : xs <- get
    put xs
    if x then inter3' a else inter3' b

go3' = evalState (inter3' test)
       [True, False]
```

Now go and read some tutorials

Practical: Why Free Monads Matter

http://www.haskellforall.com/2012/06/you-could-have-invented-free-monads.html

Theoretical: Many Roads to Free Monads

https://www.fpcomplete.com/user/dolio/many-roads-to-free-monads

I'll put this presentation and some sample code at

http://blog.sigfpe.com