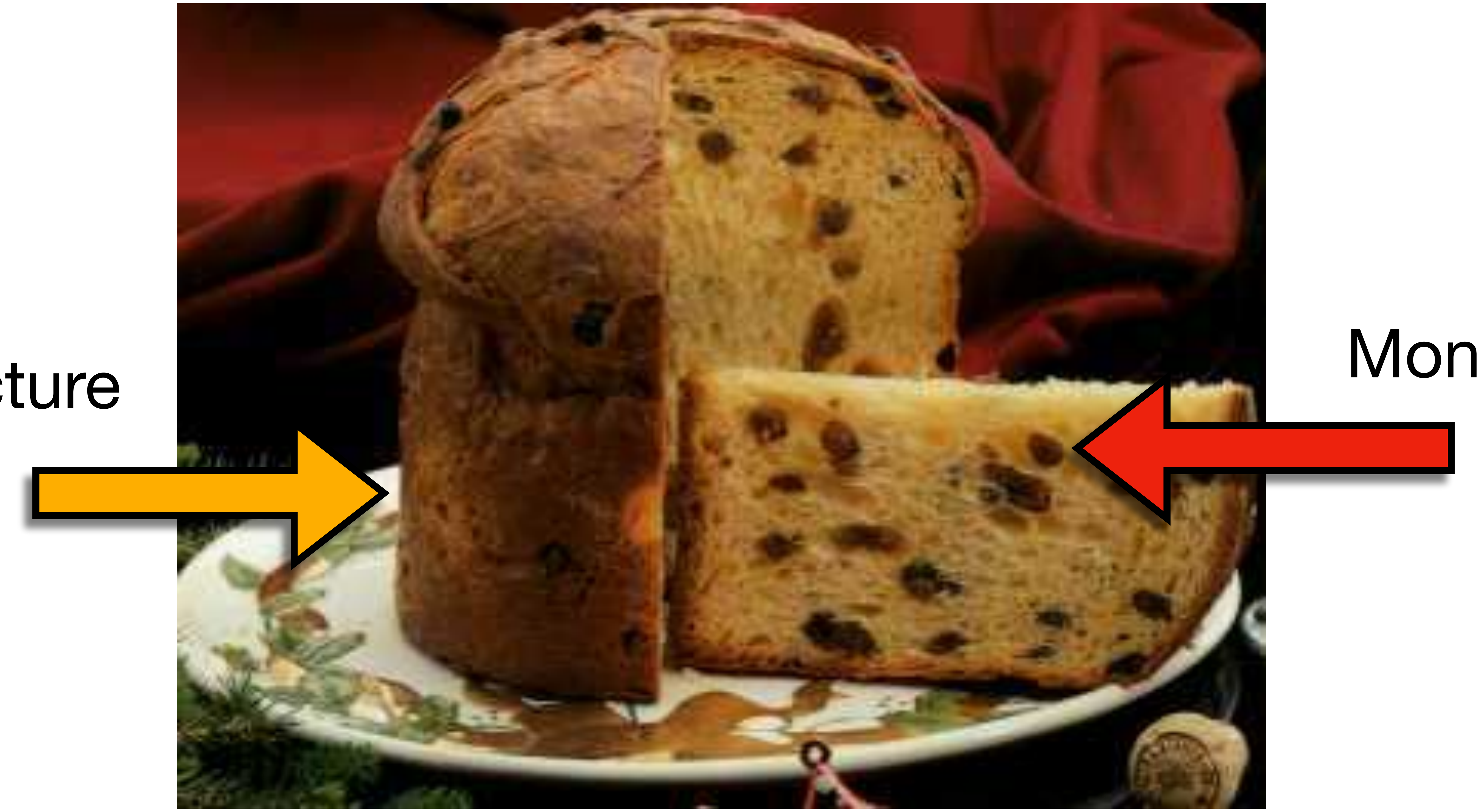
Monoidal Catamorphisms

Recursion Schemes with Monoids

Bartosz Milewski, 2020

Data Structure



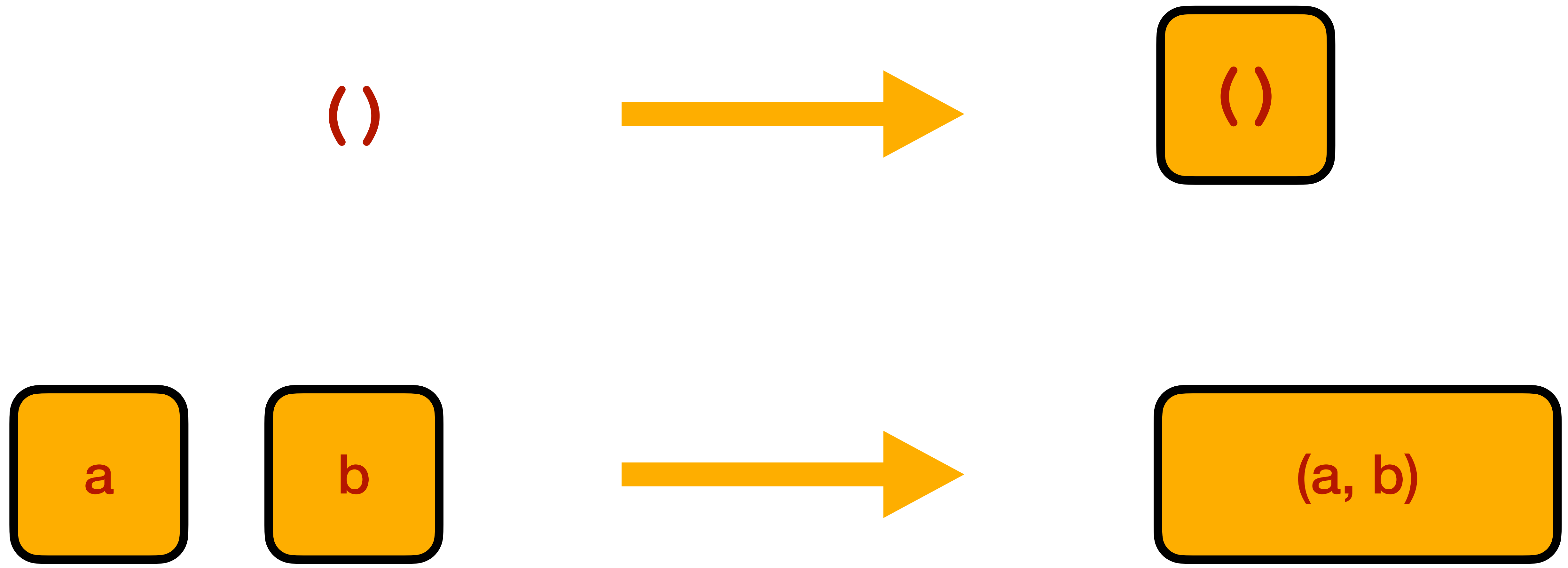
Monoidal Value

Can be easily parallelized

```haskell
data Fold a b = forall m. Monoid m => Fold (a -> m) (m -> b)
```

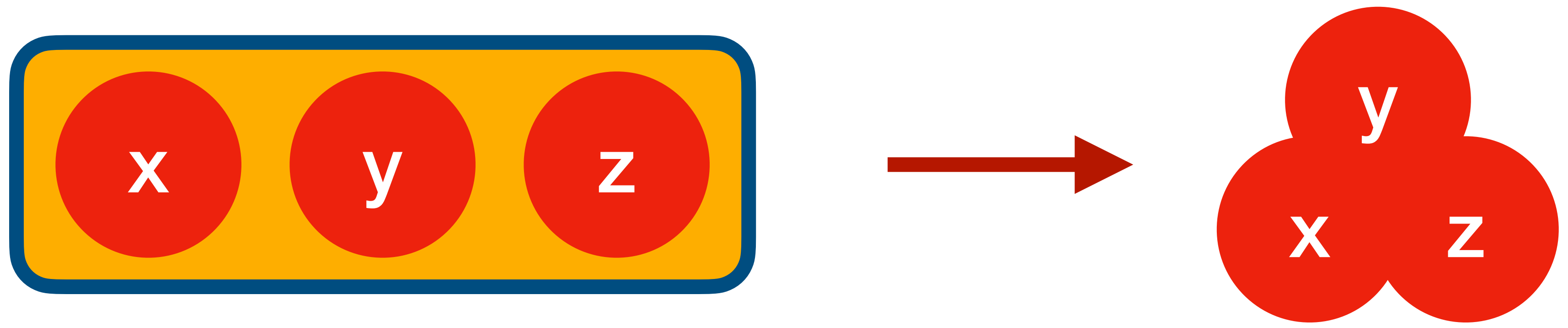
```haskell
fold :: Fold a b -> [a] -> b
fold (Fold toM fromM) = fromM . mconcat . fmap toM
```

```haskell
instance Functor (Fold a) where
  fmap f (Fold toM fromM) = Fold toM (f . fromM)


class Monoidal f where
  init    :: f ()
  combine :: f a -> f b -> f (a, b)
```

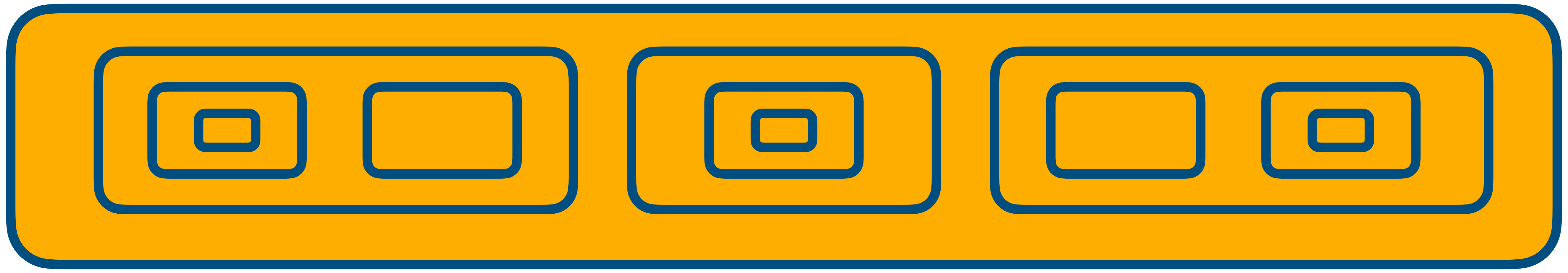


```haskell
instance Monoidal (Fold a) where
  -- Fold a ()
  init = Fold bang id
  -- Fold a b -> Fold a c -> Fold a (b, c)
  combine (Fold t f) (Fold t' f') = Fold (tuple t t') (bimap f f′)

bang :: a -> ()
bang _ = ()

tuple :: (c -> a) -> (c -> b) -> (c -> (a, b))
tuple f g = \c -> (f c, g c)
```

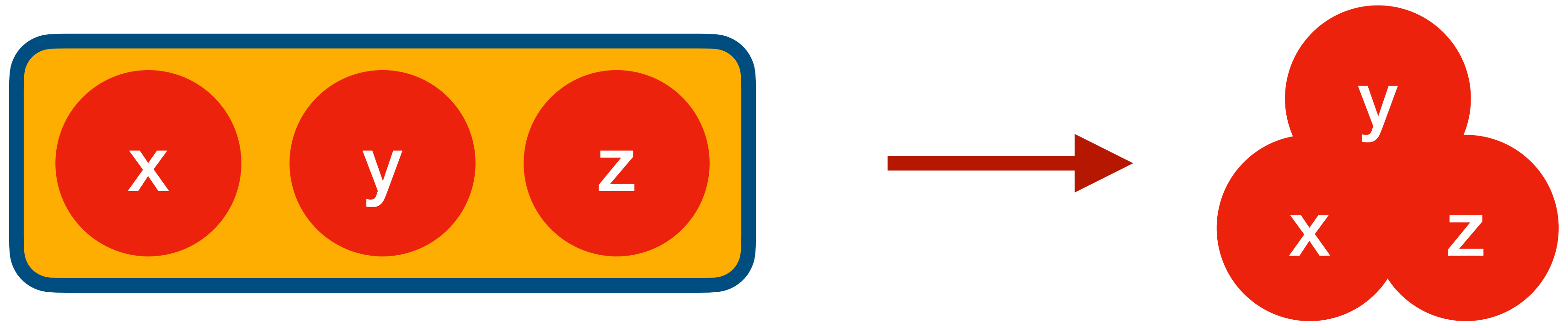
```
type Algebra f a = f a -> a
```

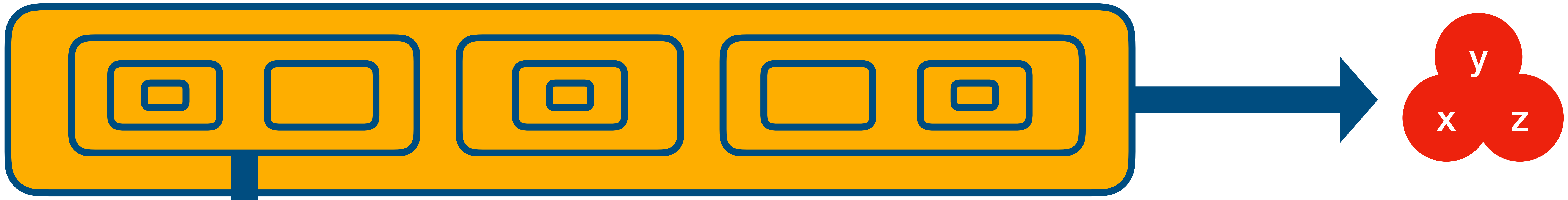



```
newtype Fix f = Fix { unFix :: f (Fix f) }
```



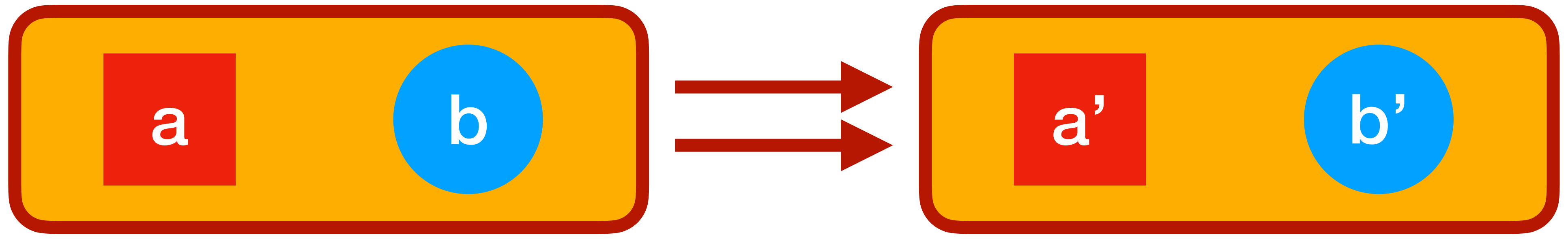
```haskell
type Algebra f a = f a -> a
```

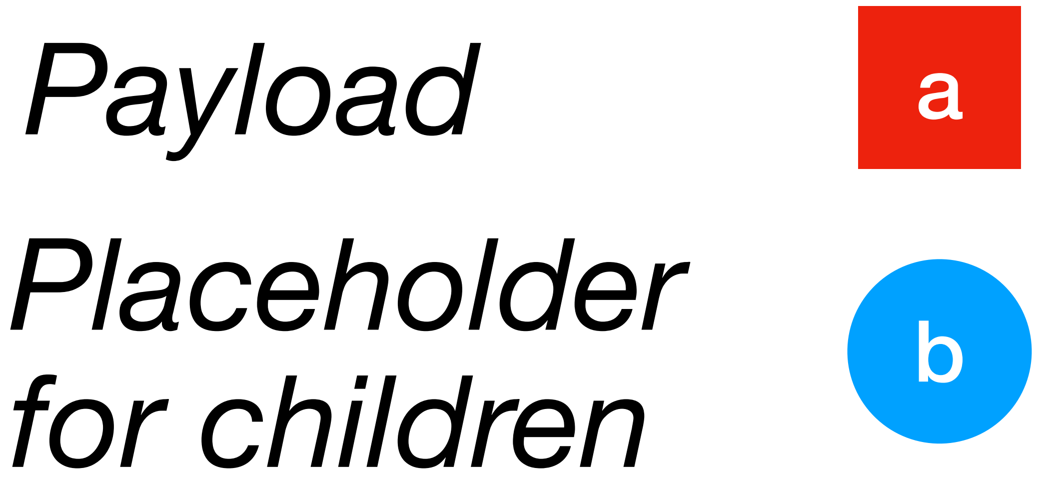




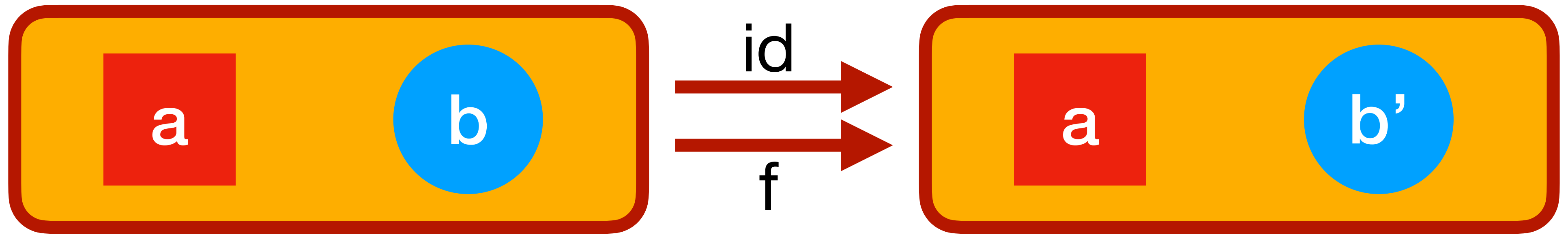

```haskell
cata :: Functor f => Algebra f a -> Fix f -> a
cata alg = alg . fmap (cata alg) . unFix
```

```haskell
class Bifunctor f where
  bimap :: (a -> a') -> (b -> b') -> f a b -> f a' b'
```

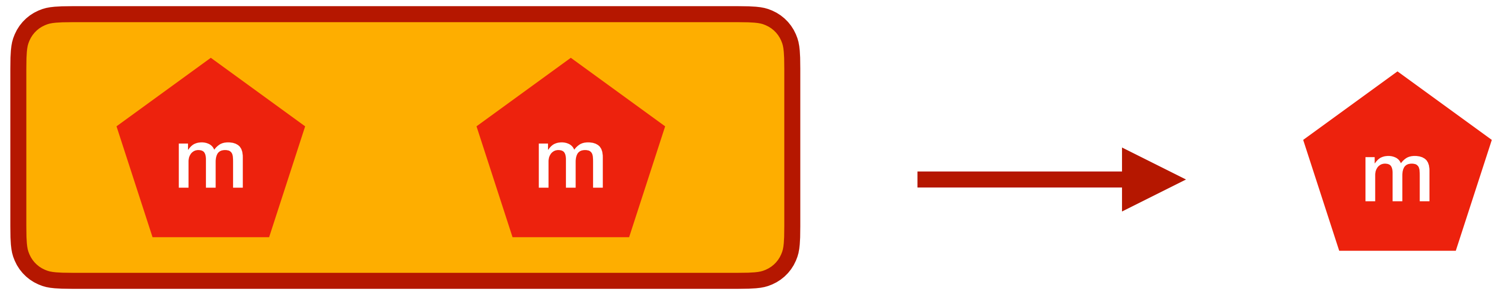




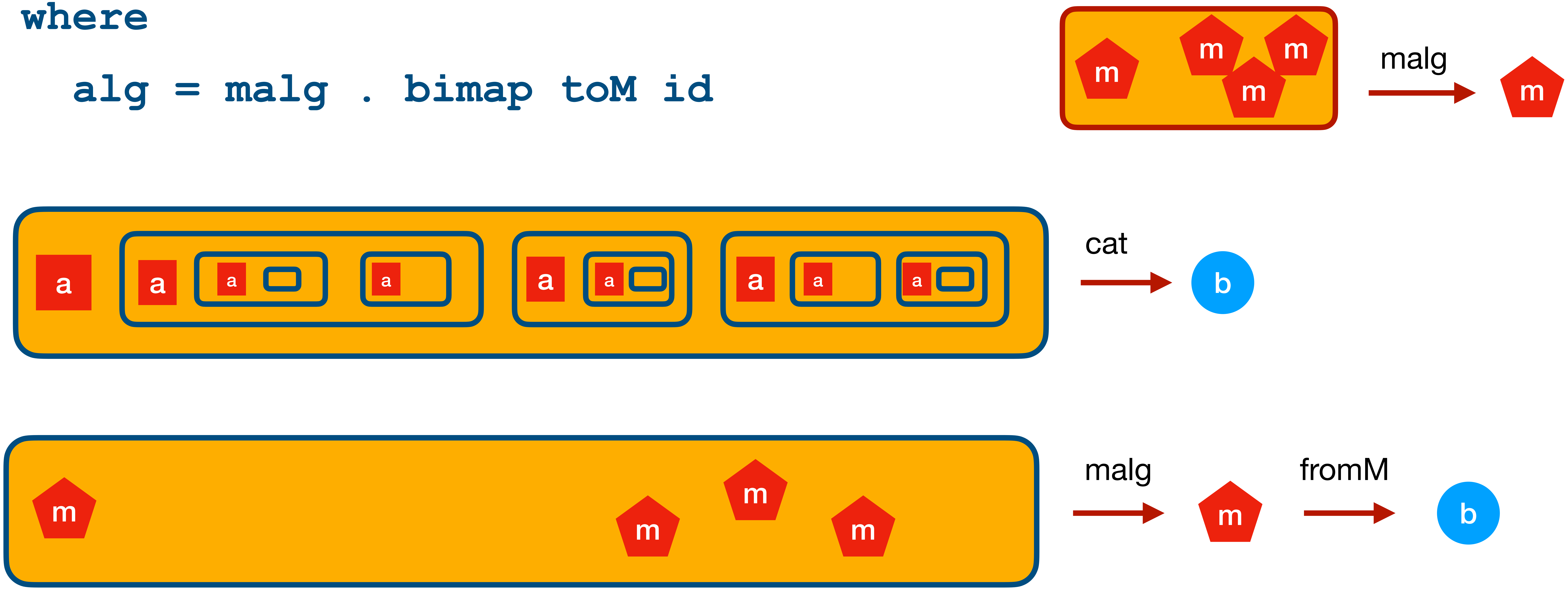

```haskell
instance Bifunctor f => Functor (f a) where
  fmap g = bimap id g
```




```haskell
type MAlgebra f = forall m. Monoid m => f m m -> m
```

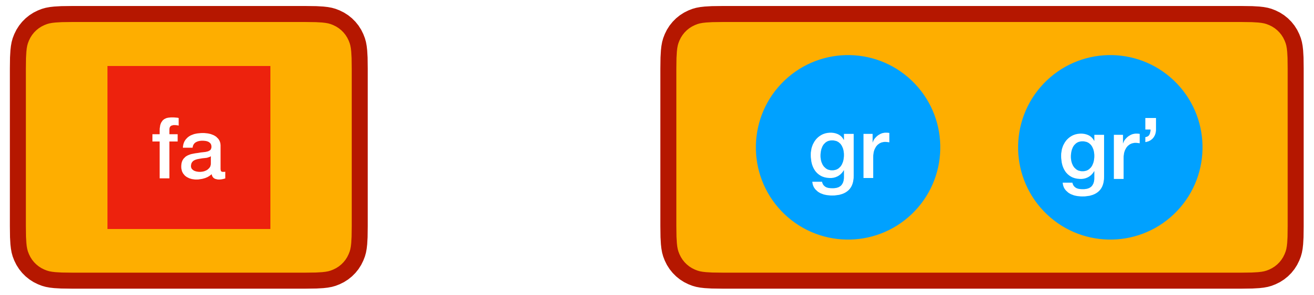


```
cat :: Bifunctor f => MAlgebra f -> Fold a b -> Fix (f a) -> b
cat malg (Fold toM fromM) = fromM . cata alg
  where
    alg = malg . bimap toM id
```

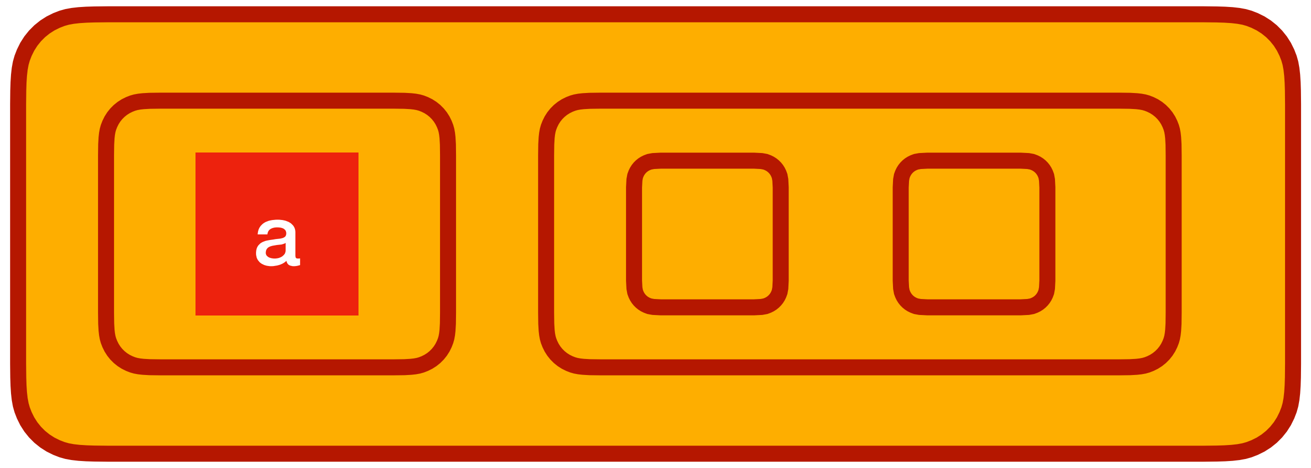


```
data TreeF a r = Leaf a | Node r r
```

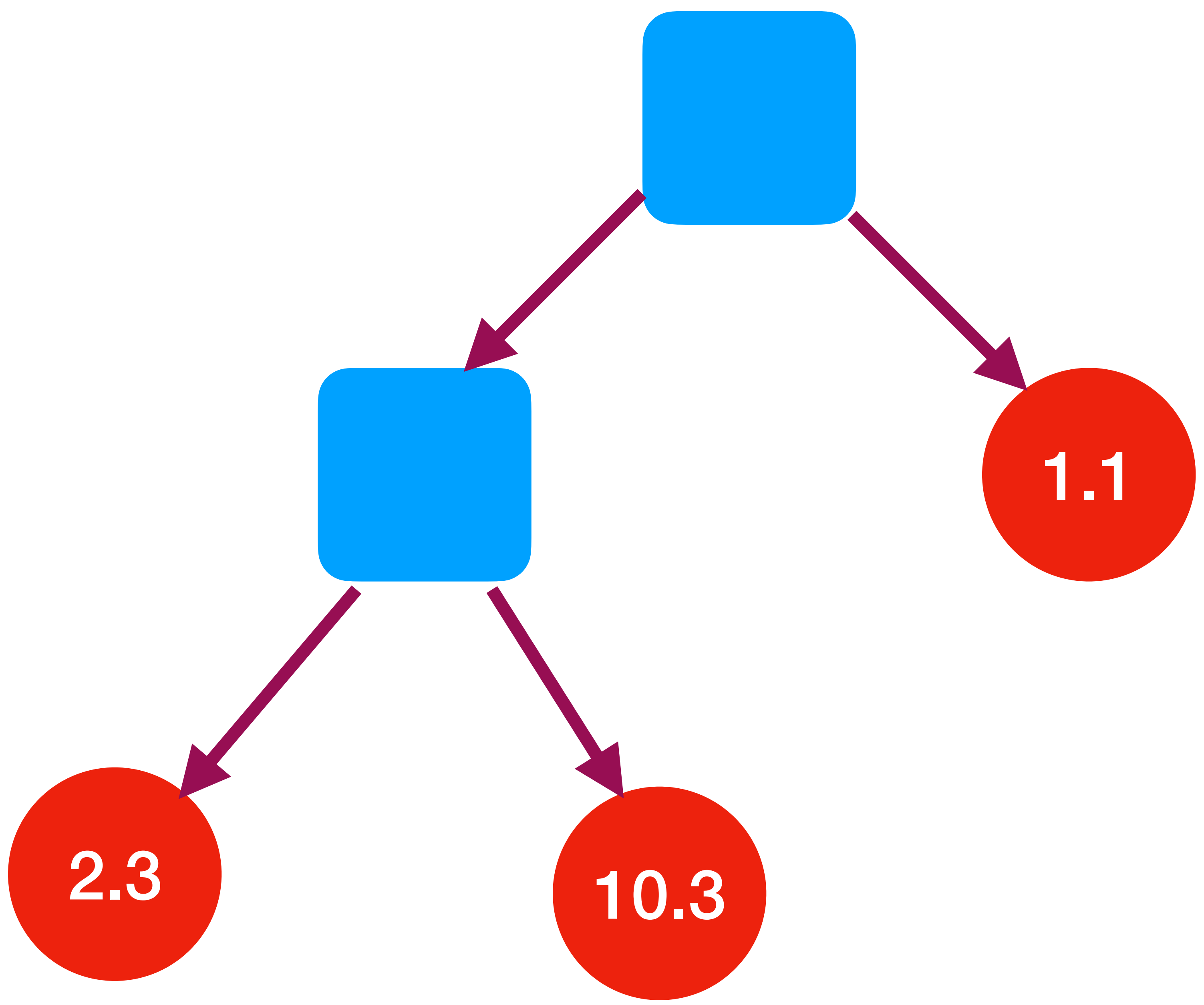

```
instance Bifunctor TreeF where
  bimap f g (Leaf a) = Leaf (f a)
  bimap f g (Node r r') = Node (g r) (g r')
```

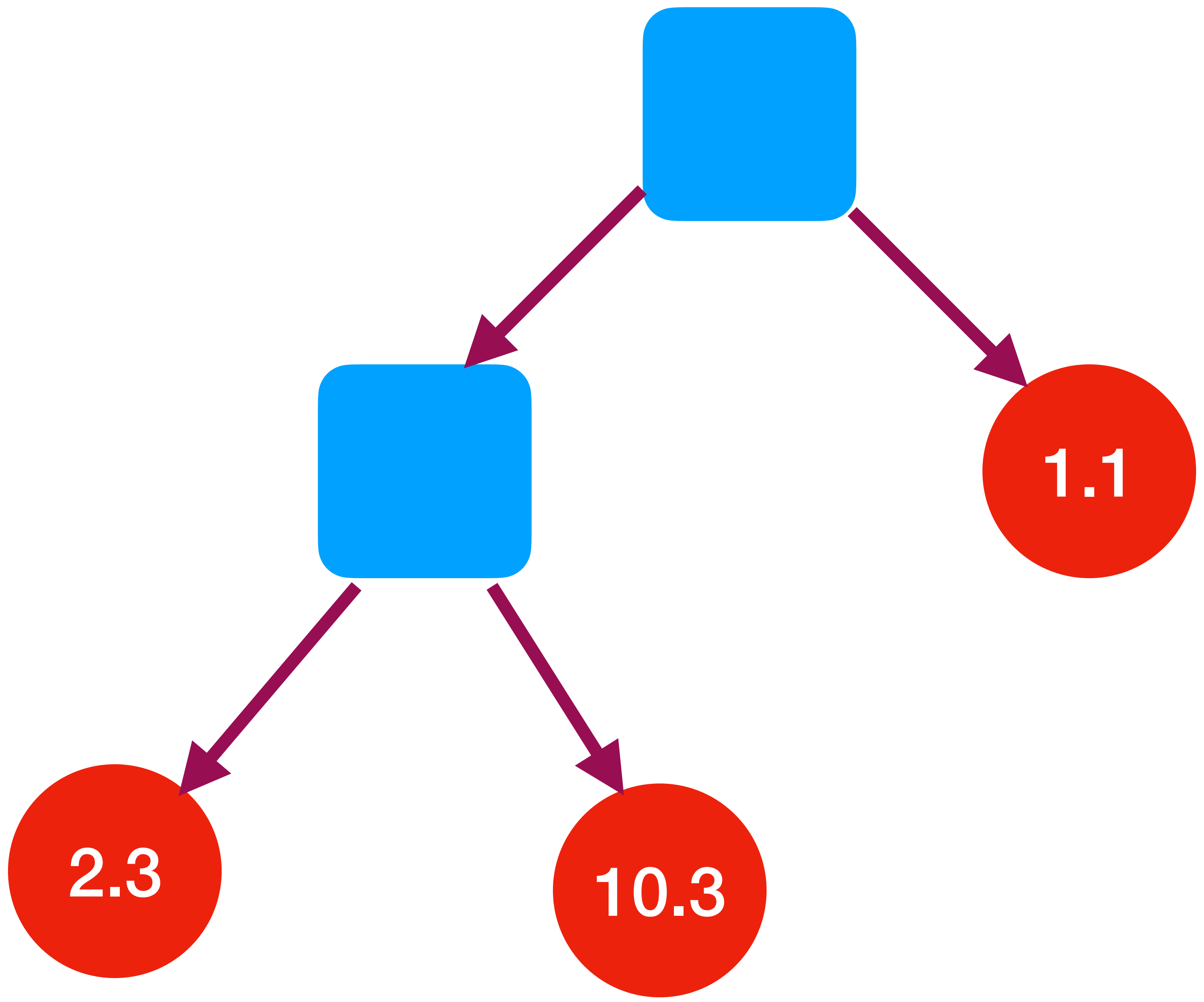



```
type Tree a = Fix (TreeF a)
```



```
leaf :: a -> Tree a
leaf a = Fix (Leaf a)

node :: Tree a -> Tree a -> Tree a
node t t' = Fix (Node t t')

myTree :: Tree Double
myTree = node (node (leaf 2.3) (leaf 10.3)) (leaf 1.1)
```






```
> cat myAlg myFold myTree
> "13"
```

```haskell
myAlg :: MAlgebra TreeF
myAlg (Leaf m) = m
myAlg (Node m m') = m <> m'


myFold :: Fold Double String
myFold = Fold floor' show'
    where
      floor' :: Double -> Sum Int
      floor' = Sum . floor
      show' :: Sum Int -> String
      show' = show . getSum
```

```
u :: a -> m
v :: n -> b
```

```
f :: m -> n
```

```
Fold (f . u) v
```

- Use the function u to extract monoidal values,
- transform these values to another monoid using f,
- do the folding in the second monoid, and
- translate the result using v

```
Fold u (v . f)
```

- Use the function u to extract monoidal values,
- do the folding in the first monoid,
- use f to transform the result to the second monoid, and
- translate the result using v