Free Categories and State Machines

Marcin Szamotulski

INPUT | OUTPUT

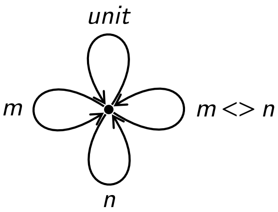
17th February 2020

Monoids

Categories with a single object

```haskell
class Semigroup s where
  -- prop> a <> (b <> c) = (a <> b) <> c
  (<>) :: s -> s -> s

class Semigroup m => Monoid m where
  -- prop> a <> unit = a = unit <> a
  unit = m

data MonoidAsCategory
       m (k :: ()) (k' :: ())
       where
  MonoidAsCategory
    :: m
    -> MonoidAsCategory m '() '()

instance Monoid m
      => Category (MonoidAsCategory m)
      where
    id :: forall m (a :: ()).
          Monoid m
       => MonoidAsCategory m a a
    id = unsafeCoerce
          (MonoidAsCategory (unit :: m))

    (MonoidAsCategory x)
       . (MonoidAsCategory y)
       = MonoidAsCategory (x <> y)
```

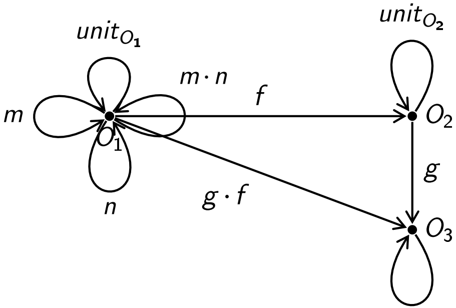
```haskell
class Category (c :: k -> k -> *) where
  -- prop> id . f = f = f . id
  id  :: c x x
  -- prop> f . (g . h) = (f . g) .h
  (.) :: c y z
      -> c x y
      -> c x z
```



Categories

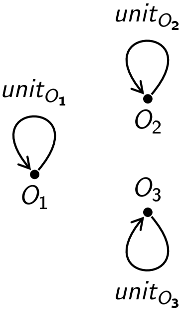
Monoids with many objects

```
class Category (c :: k -> k -> *) where
  -- prop> id . f = f = f . id
  id  :: c x x
  -- prop> f . (g . h) = (f . g) .h
  (.) :: c y z
      -> c x y
      -> c x z

instance Category c
      => Monoid (c a a)
      where
  unit = id
  x <> y = x . y
```

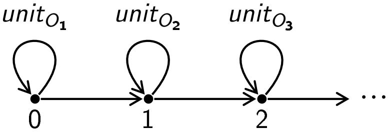


Examples

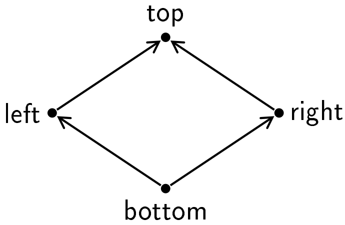
Any set is a (discrete) category



Natural numbers



Examples



Free Algebras

```haskell
type family AlgebraType  (f :: k) (a :: l) :: Constraint
type family AlgebraType0 (f :: k) (a :: l) :: Constraint

data Proof (c :: Constraint) (a :: l) where
    Proof :: c => Proof c a

class FreeAlgebra (m :: Type -> Type)  where
    returnFree :: a -> m a

    foldMapFree
        :: forall d a.
         ( AlgebraType  m d
         , AlgebraType0 m a
         )
        =>   (a -> d) -- ^ a map generators into @d@
        -> (m a -> d) -- ^ a homomorphism from @m a@ to @d@

    codom  :: forall a. AlgebraType0 m a => Proof (AlgebraType m (m a)) (m a)

    default codom :: forall a. AlgebraType m (m a)
                  => Proof (AlgebraType m (m a)) (m a)
    codom = Proof

    forget :: forall a. AlgebraType  m a => Proof (AlgebraType0 m a) (m a)

    default forget :: forall a. AlgebraType0 m a
                   => Proof (AlgebraType0 m a) (m a)
    forget = Proof
```

Free Semigroups and Monoids

```haskell
type instance AlgebraType0 NonEmpty a = ()
type instance AlgebraType  NonEmpty m = Semigroup m
instance FreeAlgebra NonEmpty where
    returnFree a = a :| []
    foldMapFree f (a :| []) = f a
    foldMapFree f (a :| (b : bs)) = f a <> foldMapFree f (b :| bs)

instance Semigroup [a] where
  -- (<>) = (++) _concatenation_
  (x : xs) <> ys = x : (xs <> ys)
  []       <> ys = ys
```

```haskell
instance Monoid [a] where
  unit = []

type instance AlgebraType [] m = Monoid m
instance FreeAlgebra [] where
    returnFree a = [a]
    -- foldMapFree = foldMap
    foldMapFree _ [] = unit
    foldMapFree f (a : as) = f a <> foldMapFree f as
```

Why Free Algebras are Important?

- A free algebra can be interpreted in any other algebra (of the same type). e.g.
  - given `f :: Monoid d => a -> d` we have a monoid homomorphism `foldMap f :: Monoid d => [a] -> d`.
  - given `f :: Semigroup d => a -> d` we can construct a semigroup homomorphism `Semigroup d => NonEmpty a -> d`
  - given `f :: Monad m => (forall x. f x -> m x)` we have monad morphism `foldFree f :: Monad m => Free f a -> m a`

Why Free Algebras are Important?

- Birhoff's Theorem!

Theorem (G.Birkhoff 1935)

Every variety is an equational theory.

- Why this is important? Because the proof constructs free algebras to show that varieties are equational theories.

- It explains *constructively why semigroups, monoids, Boolean or Heyting algebras have free algebras.*

Higher Kinded Free Algebras

```haskell
class FreeAlgebra2 (m :: (k -> k -> Type) -> k -> k -> Type) where
  liftFree2    :: f a b
               -> m f a b

  foldNatFree2 :: forall (d :: k -> k -> Type)
                         (f :: k -> k -> Type) a b .
                  ( AlgebraType  m d
                  , AlgebraType0 m f
                  )
               => (forall x y. f x y -> d x y)
               -> (m f a b -> d a b)

  codom2  :: forall (f :: k -> k -> Type).
             AlgebraType0 m f
          => Proof (AlgebraType m (m f)) (m f)

  default codom2 :: forall a. AlgebraType m (m a)
                 => Proof (AlgebraType m (m a)) (m a)
  codom2 = Proof

  forget2 :: forall (f :: k -> k -> Type).
             AlgebraType m f
          => Proof (AlgebraType0 m f) (m f)

  default forget2 :: forall a. AlgebraType0 m a
                  => Proof (AlgebraType0 m a) (m a)
  forget2 = Proof
```

Free Categories

type aligned sequences

```
data ListTr :: (k -> k -> *) -> k -> k -> * where
  NilTr  :: ListTr f a a
  ConsTr :: f b c -> ListTr f a b -> ListTr f a c

instance Category ListTr where
  id = NilTr
  -- concatenation
  (ConsTr x xs) . ys = ConsTr x (xs . ys)
  NilTr         . ys = ys

type instance AlgebraType  ListTr c = Category c
instance FreeAlgebra2 ListTr where
  liftFree2 a  = ListTr a NilTr
  foldNatFree2 _   NilTr          = id
  foldNatFree2 fun (ConsTr bc ab) = fun bc . foldNatFree2 fun ab
```

Free Categories

type aligned sequences

```
data ListTr :: (k -> k -> *) -> k -> k -> * where
  NilTr  :: ListTr f a a
  ConsTr :: f b c -> ListTr f a b -> ListTr f a c

instance Category ListTr where
  id = NilTr
  -- concatenation
  (ConsTr x xs) . ys = ConsTr x (xs . ys)
  NilTr         . ys = ys

type instance AlgebraType  ListTr c = Category c
instance FreeAlgebra2 ListTr where
  liftFree2 a  = ListTr a NilTr
  foldNatFree2 _   NilTr          = id
  foldNatFree2 fun (ConsTr bc ab) = fun bc . foldNatFree2 fun ab
```

There are other possible representations:

- Okasaki's realtime queues
- Church encoding
- ...

Benchmarks: `https://coot.me/bench-cats.html`

Kleisli categories

… and beyond

```haskell
newtype Kleisli m a b = Kleisli { runKleisli :: a -> m b }

instance Monad m => Category (Kleisli m) where
    id = Kleisli return
    -- (.) ~ (>=>)
    (Kleisli f) . (Kleisli g) = Kleisli (\x -> g x >>= f)
```

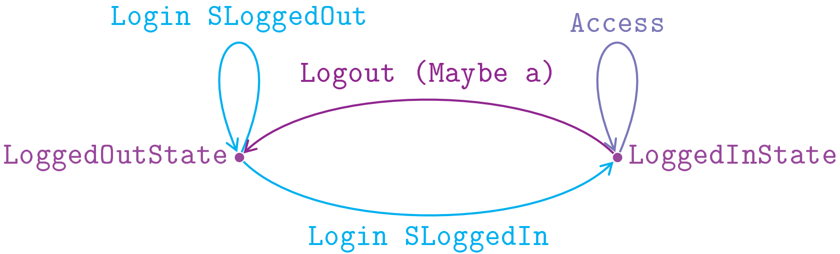
Effectful Categories

```haskell
-- | Categories which can lift monadic actions.
--
class Category c => EffectCategory c m | c -> m where
  effect :: m (c a b) -> c a b

instance Monad m => EffectCategory (Kleisli m) m where
  effect m = Kleisli (\a -> m >>= \(Kleisli f) -> f a)

instance EffectCategory (->) Identity where
  effect = runIdentity
```

Free Effectful Categories

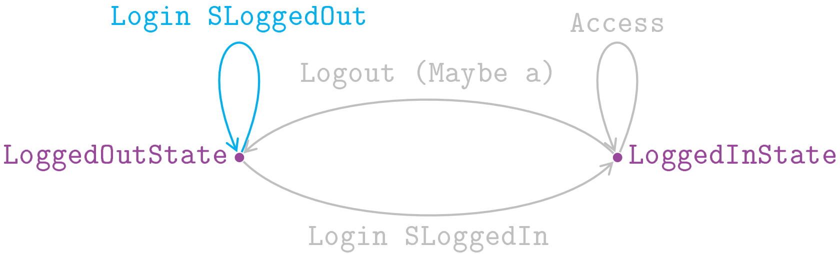
```haskell
-- | Category transformer, which adds @'EffectCategory'@ instance to the
-- underlying base category.
--
data EffCat :: (* -> *) -> (k -> k -> *) -> k -> k -> * where
  Base   :: c a b -> EffCat m c a b
  Effect :: m (EffCat m c a b) -> EffCat m c a b

instance (Functor m, Category c) => Category (EffCat m c) where
  id = Base id
  Base f    . Base g    = Base    (f . g)
  f         . Effect mg = Effect ((f .) <$> mg)
  Effect mf . g         = Effect ((. g) <$> mf)

instance (Functor m, Category c) => EffectCategory (EffCat m c) m where
  effect = Effect

type instance AlgebraType0 (EffCat m) c = (Monad m, Category c)
type instance AlgebraType  (EffCat m) c  = EffectCategory c m
instance Monad m => FreeAlgebra2 (EffCat m) where
  liftFree2 = Base
  foldNatFree2 nat (Base cab)    = nat cab
  foldNatFree2 nat (Effect mcab) = effect (foldNatFree2 nat <$> mcab)
```

Example

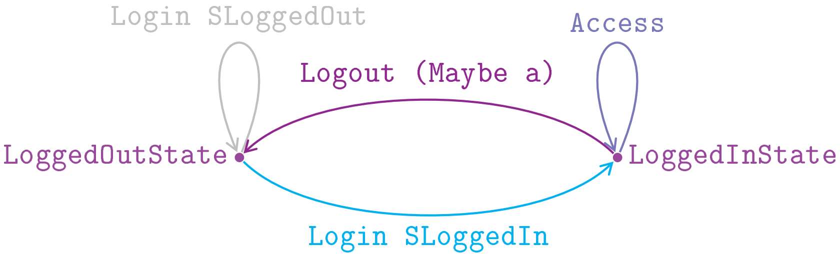


Login SLoggedOut
Access
Logout (Maybe a)
LoggedOutState
LoggedInState
Login SLoggedIn

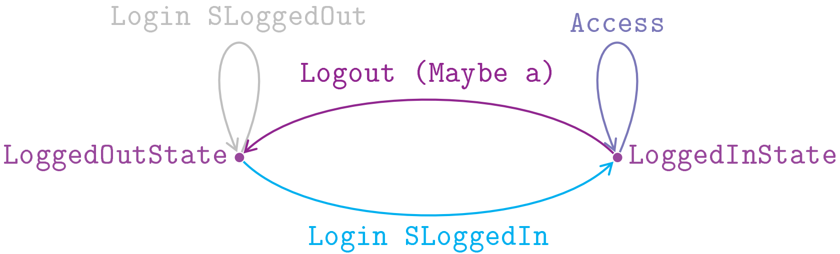
Example


Login SLoggedOut
Access
Logout (Maybe a)
LoggedOutState
LoggedInState
Login SLoggedIn

Example


Login SLoggedOut
Access
Logout (Maybe a)
LoggedOutState
LoggedInState
Login SLoggedIn

Example



`https://github.com/coot/free-category`
(examples directory)

Thank You!