

Kind polymorphism

by Krzysztof Gogolewski (krz.gogolewski@gmail.com)

October 26, 2011

Draft version. This article presents an application of polymorphism on kind level.

Consider the following type class.

```
class Function f x where
  val :: f x
```

```
instance Function Maybe Int where
  val = Just 5
```

An instance of `Function f x` consists of a value of type `f x`. In the first instance, `f = Maybe :: * -> *` and `x = Int :: *`. But the type expression `f x` is well-typed not only with those kinds. You could also attempt to write:

```
data Apply f = Apply (f Int)
```

```
instance Function Apply Maybe where
  val = Apply (Just 5)
```

since `Apply (Just 5)` is a fine value of type `Apply Maybe`.

However, GHC won't accept this definition. It requires that `f` and `x` have simplest possible kinds – in this case, `* -> *` and `*`. It is possible to override this inference and create a separate type class for another kind:

```
class Function2 (f :: (* -> *) -> *) x where
  val2 :: f x
```

```
instance Function2 Apply Maybe where
  val2 = Apply (Just 5)
```

We now have two classes, both having the same functionality – type application `f x`.

Kind polymorphism is a language feature allowing to make a single definition that can be used with different kinds. Without kind polymorphism, we have to write a redundant type class `Function2` and add more type classes if other kinds are demanded.

This example is artificial. In this article, I will show you a construct which can be used with many different kinds, corresponding to several known Haskell structures.

I assume the reader has a good understanding of Haskell type classes like `monoid`, `monad`, `category`. For an overview, see Brent Yorgey’s *Typeclassopedia* [1]. I also assume the reader knows rank-2-types and existential types.

Using UHC

Kind polymorphism isn’t supported in the prevailing Haskell compiler, GHC, as of version 7.2. There is a compiler called *Utrecht Haskell Compiler* [2] which supports kind polymorphism. It compiles successfully instances of both `Function Maybe Int` and `Function Apply Maybe` with no modifications.

I will proceed to a first version of a multifunctional type class.

```
class Monoid mor unit prod m where
  id :: mor unit m
  mult :: mor (prod m m) m
```

An instance of `Monoid mor unit prod m` consists of two values, one of type `mor unit m` and second of type `mor (prod m m) m`. Remember those two types – they will be used often.

Instances of `Monoid (->) () (,) m` are objects with functions `() -> m` and `(m,m) -> m`.

We can define an instance of this class:

```
instance Monoid (->) () (,) Integer where
  id () = 0
  mult (a,b) = a + b
```

This is Haskell’s `instance Monoid (Sum Integer)`.

The implicit assumption is that `mult` is associative and `id` is its identity element. Later, we will see it in general form.

It turns out, that the same type class, used with different kinds, gives the monad interface.

In Haskell, monads are constructors `m` with functions

- ▶ `return :: a -> m a`
- ▶ `(>>=) :: m a -> (a -> m b) -> m b`

An equivalent formulation is that a monad is a constructor `m` with functions

- ▶ `fmap :: (a -> b) -> m a -> m b`
- ▶ `return :: a -> m a`
- ▶ `join :: m (m a) -> m a`

You can check the equivalence by writing `(>>=)` in terms of `fmap` and `join`, and `fmap` and `join` in terms of `return` and `(>>=)`.

To write the monad class we will have to use data declarations (or newtypes) to get the types in desired form `mor unit m` and `mor (prod m m) m`. From now on, I will often call types “equivalent” when they can be transformed by adding or removing some constructors. After defining

```
data Id x = Id x
data Compose f g x = Compose (f (g x))
```

the function `return` can be written as `Id a -> m a` and `join` as `Compose m m a -> m a`. If we define also:

```
data NatT f g = NatT (forall a. f a -> g a)
```

we can rewrite `return` as `NatT Id m` and `join` as `NatT (Compose m m) m`. These are exactly types in `Monoid` type class, with `mor = NatT`, `unit = Id` and `prod = Compose`.

Below is monad instance for `Maybe`:

```
instance Monoid NatT Compose Id Maybe where
  one = NatT $ \(Id x) -> Just x
  mult = NatT $ \(Comp x) -> join x
  where join (Just (Just x)) = Just x
        join _ = Nothing
```

One of the parts of a monad, `fmap`, is missing here. I will address this issue later, after we will get comfortable.

We have unified notions of monoid and monad, even though they have different kinds. The three objects `mor`, `unit`, `prod` determine what is the entity the `Monoid` class represents. Its worth to compare the kinds:

	monoid	monad
<code>mor</code>	<code>(->) :: * -> * -> *</code>	<code>NatT :: (* -> *) -> (* -> *) -> *</code>
<code>unit</code>	<code>() :: *</code>	<code>Id :: * -> *</code>
<code>prod</code>	<code>(,) :: * -> * -> *</code>	<code>Compose :: (* -> *) -> (* -> *) -> * -> *</code>
<code>m</code>	<code>Int :: *</code>	<code>Maybe :: * -> *</code>

The Monad.Reader

Generally, we can write the kinds as:

```
mor :: k -> k -> *
unit :: k
prod :: k -> k -> k
m :: k
```

where `k` is either `*` for monoids, or `* -> *` for monads. I hope you see `mor unit m` and `mor (prod m m) m` are valid types of kind `*`.

After discovering this I started to wonder if there are other examples. It turns out that there are.

Below is a listing containing some of the type classes. Notice each of them has two members, one for identity and one for multiplication.

```
class IxFunctor f where                                -- Auxiliary
  imap :: (a -> b) -> f s s' a -> f s s' b
```

```
class Monoid a where
  mempty  :: a
  mappend :: a -> a -> a
```

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

```
class Functor w => Comonad w where
  extract :: w a -> a
  duplicate :: w (w a) -> w a
```

```
class Category c where
  id :: c x x
  (.) :: c y z -> c x y -> c x z
```

```
class IxFunctor m => IxMonad m where
  ireturn :: a -> m s s a
  ijoin   :: m s1 s2 (m s2 s3 a) -> m s1 s3 a
```

```
class AbelianGroup m => Ring m where
  one :: m
  mult :: m -> m -> m
```

However, expressing them using kind polymorphism requires GADTs. Since currently GHC doesn't support kind polymorphism, and UHC doesn't support GADTs, it seems that we are stuck.

Introducing Omega

Fortunately, there is a language which supports both of the features, called Omega [3] (often stylized as Ω mega). It is close enough to Haskell. Let me warn you:

- ▶ Omega is an experimental project. I stumbled on a few implementation bugs. Error messages when playing with types are not easy to understand, even if you have experience with somewhat distinctive Haskell errors.
- ▶ Omega doesn't have typeclasses. We will express `Monoid` using `data`.
- ▶ Unlike Haskell, it's strict. However, I won't rely on strictness or laziness anywhere in this article.
- ▶ Its standard Prelude contains only a handful of functions. There is no `Integer`, only `Int`.
- ▶ Omega uses `->` arrow between types (for example, `Int -> Int`), but `~>` arrow between kinds (for example, `* ~> *`)

Omega allows to define new kinds just like Haskell allows to define new types, using `kind` instead of `data`. Below is a type-level list.

```
kind List a = Nil | Cons a (List a)

type List1 = Cons Int (Cons Bool Nil) -- :: List *
type List2 = Cons Maybe Nil           -- :: List (* ~> *)
-- pseudocode version:
-- type List1 = [Int, Bool] :: [*]
-- type List2 = [Maybe] :: [* ~> *]
```

Just like elements of a Haskell list have to be the same type, elements of a type-level list must be of the same kind.

In the monoid/monad example, the type of structure was indicated by `mor`, `unit`, `prod`. We will group them into a tuple. These are the general kinds that were deducted before - `mor`, `unit` and `prod`.

```
kind MCategory k = MCat (k ~> k ~> *) k (k ~> k ~> k)
```

The name of this tuple stems from “monoidal category”, which is a framework where monoids can be considered.

Here's a type of kind `MCategory *`, which was used in the monoid example.

```
type Hask = MCat (->) () (,)
```

Here is the crucial `Monoid` type translated to Omega. It is exactly the same as UHC's `Monoid` type class, but with `mor`, `unit`, `prod` packed into the tuple.

Initially getting those four lines right took me over three hours. Don't get frustrated.

```
data Monoid :: forall (k :: *1). MCategory k ~> k ~> *0 where
  M :: forall (k :: *1) (m :: k) mor unit prod.
      mor unit m -> mor (prod m m) m ->
      Monoid (MCat mor unit prod) m
```

- ▶ Since Omega doesn't support type classes, I converted `class` to a GADT. In Haskellish pseudosyntax, that would be:

```
data Monoid (MCat mor unit prod) m where
  M :: mor unit m -> mor (prod m m) -> Monoid (MCat mor unit prod) m
```

- ▶ `k :: *1` means that `k` is a kind. Here, `*1` is the "type" of kinds. Omega has an infinite hierarchy that divides every entity into one of the levels. Values have types, types have kinds, kinds have sorts etc. Everything is classified by something that is situated one level up. `*n` is typed `*(n+1)`. In fact, `*` is a synonym for `*0`.
- ▶ The type of `M` is the following: For any kind `k` and type `m` of kind `k`, and types `mor`, `unit`, `prod`, two values of type `mor unit m` and `mor (prod m m) m` are arguments to constructor for `Monoid (MCat mor unit prod) m`.

Monoids

Let's start with monoids. This code is the same as monoids in UHC.

```
sumMonoid :: Monoid Hask Int
sumMonoid = M (\() -> 0) (\(x,y) -> x+y)
```

Given two monoids, you can form a **product monoid**, which is a monoid of pairs.

```
prodmon :: Monoid Hask m -> Monoid Hask n -> Monoid Hask (m,n)
prodmon (M unit1 mult1) (M unit2 mult2) =
  M (\() -> (unit1 (), unit2 ()))
    (\((a,b),(a',b')) -> (mult1 (a,a'), mult2 (b,b')))
```

Monads

Here's the monad instance for `Maybe` in `Omega`:

```
data NatT f g = NatT (forall x. f x -> g x)
data Id x = Id x
data Compose f g x = Compose (f (g x))

type EndHask = MCat NatT Id Compose -- :: MCategory (*0 ~> *0)

maybeMonad :: Monoid EndHask Maybe
maybeMonad = M (NatT (\(Id x) -> Just x)) (NatT join)
  where join (Compose (Just (Just x))) = Just x
        join _ = Nothing
```

The `writer` monad can transform any monoid to a monad.

```
data Writer m a = Writer m a

writer :: Monoid Hask m -> Monoid EndHask (Writer m)
writer (M unit mult) = M (NatT ret) (NatT join)
  where ret (Id a) = Writer (unit ()) a
        join (Compose (Writer x (Writer y z))) = Writer (mult (x,y)) z
```

Comonads

A comonad is a type constructor `w` with the following functions:

- ▶ `fmap :: (a -> b) -> w a -> w b`
- ▶ `extract :: w a -> a`
- ▶ `duplicate :: w a -> w (w a)`

which are the same as the monad's, but with arrows going the other way.

```
data Dual c a b = Dual (c b a)

type CoEndHask = MCat (Dual NatT) Id Compose
```

Below is the product comonad (also known as environment or coreader comonad):

```
data Env s a = Env s a

envComonad :: Monoid CoEndHask (Env s)
envComonad = M extract duplicate
```

The Monad.Reader

```
where extract :: Dual NatT Id (Env s)
      extract = Dual (NatT (\(Env s a) -> Id a))

      duplicate :: Dual NatT (Compose (Env s) (Env s)) (Env s)
      duplicate = Dual (NatT (\(Env s a) -> Compose (Env s (Env s a))))
```

Below is the store comonad (also known as costate comonad).

```
data Store s a = Store (s -> a) s

storeComonad :: Monoid CoEndHask (Store s)
storeComonad = M extract duplicate
  where extract :: Dual NatT Id (Store s)
        extract = Dual (NatT (\(Store f x) -> Id (f x)))

        duplicate :: Dual NatT (Compose (Store s) (Store s)) (Store s)
        duplicate = Dual (NatT (\(Store f x) -> Compose (Store (Store f) x)))
```

Categories

A category consists of a binary type constructor c and functions

- ▶ $\text{id} :: c\ x\ x$
- ▶ $(.) :: c\ y\ z \rightarrow c\ x\ y \rightarrow c\ x\ z$

One might observe identity vaguely resembles identity of a monoid, and composition resembles monoid multiplication. This is the case, but requires a little rewriting. A category is a graph whose edges may be multiplied.

Omega has the following type built-in. It is used to mark types as equal.

```
data Equal a b where
  Eq :: Equal a a
```

Observe that $\text{id} :: c\ x\ x$ is equivalent to $\text{id} :: \text{Equal}\ x\ y \rightarrow c\ x\ y$. If we define

```
data GraT c d = forall x y. c x y -> d x y
```

we can write id as $\text{GraT}\ \text{Equal}\ c$, which fits the general `Monoid` definition for $\text{mor} = \text{GraT}$ and $\text{unit} = \text{Equal}$.

Let's do the same for $(.)$. If we define

```
data GCompose g h x z = forall y. GCompose (g x y) (h y z)
```

then $(.)$'s type is equivalent to $\text{GCompose}\ c\ c\ x\ z \rightarrow c\ x\ z$ and then can be written as $\text{GraT}\ (\text{GCompose}\ c\ c)\ c$. Notice existential type.

We can now use `Monoid` to define a category. This time, the kind is $* \sim > * \sim > *$.

```

type Graph = MCat GraT Equal GCompose

haskCat :: Monoid Graph (->)
haskCat = M (GraT ident) (GraT compose)
  where ident :: Equal a b -> a -> b
        ident Eq x = x
        compose (GCompose g h) = g . h

```

Above, we have to specify the type of `ident`. Look at this line in isolation:

```
ident Eq x = x
```

We have to specify the signature `Equal a b -> a -> b`. Otherwise, Omega will infer `Equal a b -> c -> c`, which is also correct, but not what is needed. As you see, with GADTs sometimes a definition might have incomparable types, and no most general one.

This is the discrete category.

```

discCat :: Monoid Graph Equal
discCat = M (GraT id) (GraT compose)
  where compose :: GCompose Equal Equal a b -> Equal a b
        compose (GCompose Eq Eq) = Eq

```

If you have a category `c`, you can create a new category where `d x y` is `c y x`. This is known as the dual category.

```

dualcat :: Monoid Graph c -> Monoid Graph (Dual c)
dualcat (M (GraT ident) (GraT compose)) = M (GraT (dualid ident))
                                           (GraT (dualcmp compose))

where
  dualid :: (forall x1 y1. Equal x1 y1 -> c x1 y1) ->
            Equal x y -> Dual c x y
  dualid ident Eq = Dual (ident Eq)

  dualcmp :: (forall x1 y1. GCompose c c x1 y1 -> c x1 y1) ->
            GCompose (Dual c) (Dual c) x y -> Dual c x y
  dualcmp compose (GCompose (Dual f) (Dual g)) =
    Dual (compose (GCompose g f))

```

Did you notice that `Dual` is kind-polymorphic? I didn't give it a kind. It was used to swap arguments of `NatT :: (* ~> *) ~> (* ~> *) ~> *` and here it is used to swap arguments of `c :: * ~> * ~> *`.

An endomorphism is a morphism $c \times x$. If you fix an object in a category and restrict `id` and `(.)` to it, you'll get a monoid with identity `id :: c x x` and multiplication `(.) :: c x x -> c x x -> c x x`. GHC has a wrapper called `Endo` (which does this for `c = (->)` only), in the library `Data.Monoid`.

```
end :: Monoid Graph m -> Monoid Hask (m x x)
end (M (GraT ident) (GraT compose)) =
  M (\() -> ident Eq)
    (\(x,y) -> compose (GCompose x y))
```

Indexed monads

Indexed monads are a generalization of monads. For an introduction, see Dan Piponi's blog post [4].

An indexed monad `m` is a constructor of kind `* ~> * ~> * ~> *` with functions:

- ▶ `return :: a -> m s s a`
- ▶ `(>>=) :: m s1 s2 a -> (a -> m s2 s3 b) -> m s1 s3 b`

or, equivalently:

- ▶ `fmap :: (a -> b) -> m s1 s2 a -> m s1 s2 b`
- ▶ `return :: a -> m s s a`
- ▶ `join :: m s1 s2 (m s2 s3 a) -> m s1 s3 a`

We can rewrite `return` and `join` (leaving `fmap` again for later) using the following declarations:

```
data IndId :: *0 ~> *0 ~> *0 ~> *0 where
  IndId :: x -> IndId s s x
```

```
data IndT m n = IndT (forall s1 s2 a. m s1 s2 a -> n s1 s2 a)
```

```
data ICompose m n s1 s3 x = forall s2. ICompose (m s1 s2 (n s2 s3 x))
```

Notice `IndId s t a` is equivalent to `pair (Equal s t, a)`, and `ICompose` uses existential quantification.

We can now verify that `IndT IndId m` is a type equivalent to indexed monad's `return` type:

```
f :: IndT IndId m
f :: forall s1 s2 a. IndId s1 s2 a -> m s1 s2 a
f :: forall s1 s2 a. (Equal s1 s2, a) -> m s1 s2 a
f :: forall s a. a -> m s s a
```

and similarly for `IndT (ICompose m m) m`:

```

g :: IndT (ICompose m m) m
g :: forall s1 s2 a. (ICompose m m) s1 s2 a -> m s1 s2 a
g :: forall s1 s2 a. (exists s. m s1 s (m s s2 a)) -> m s1 s2 a
g :: forall s1 s2 s a. m s1 s (m s s2 a) -> m s1 s2 a

```

Here is the indexed state monad, written using the Monoid data type:

```

data IxState s1 s2 a = IxState (s1 -> (a, s2))
runIxState (IxState x) = x

ixstate :: Monoid Indexed IxState
ixstate = M (IndT ret) (IndT join)
  where ret :: IndId s s' x -> IxState s s' x
        ret (IndId x) = IxState (\s -> (x,s))
        join (ICompose x) = IxState (\s -> let (y, s') = runIxState x s
                                           in runIxState y s')

```

The `writer` function, giving a monad out of a monoid, has a corresponding function for indexed monads, that forms an indexed monad out of a category.

```

data IxWriter c x y a = IxWriter (c x y) a

ixwriter :: Monoid Graph c -> Monoid Indexed (IxWriter c)
ixwriter (M id comp) = M (IndT (ret id)) (IndT (join comp))
  where ret :: GraT Equal c -> IndId u v a -> IxWriter c u v a
        ret (GraT i) (IndId a) = IxWriter (i Eq) a

  join :: GraT (GCompose c c) c ->
        ICompose (IxWriter c) (IxWriter c) x y a -> IxWriter c x y a
  join (GraT c) (ICompose (IxWriter f (IxWriter g x))) =
        IxWriter (c (GCompose f g)) x

```

What happens when you fix `s` and consider `IxState s s`? This is the same as `State s` monad. Just like endomorphisms in a category form a monoid, “endomorphisms” in an indexed monad form a monad.

```

end' :: Monoid Indexed m -> Monoid EndHask (m x x)
end' (M (IndT ident) (IndT compose)) =
  M (NatT (\(Id x) -> ident (IndId x)))
    (NatT (\(Compose x) -> compose (ICompose x)))

```

Indexed comonads

Just like state with changing type is an indexed monad, store with changing type is an indexed comonad.

```
data IxStore s t a = IxStore (t -> a) s
```

An indexed comonad is a type constructor w of kind $* \sim > * \sim > * \sim > *$ with the following functions:

- ▶ `fmap :: (a -> b) -> w s1 s2 a -> w s1 s2 b`
- ▶ `extract :: w s s a -> a`
- ▶ `duplicate :: w s1 s3 a -> w s1 s2 (w s2 s3 a)`

which are the same as the indexed monad's, but with arrows going the other way.

We can reuse the `Dual` constructor, this time to flip arguments to `IndT :: (* ~> * ~> *) ~> (* ~> * ~> *) ~> *`.

Let's check if `extract` type can be modelled for `unit = IndI` and `mor = Dual IndT`.

```
extract :: Dual IndT IndId m
extract :: IndT m IndId
extract :: forall s1 s2 a. m s1 s2 a -> IndId s1 s2 a
extract :: forall s1 s2 a. m s1 s2 a -> (Equal s1 s2, a)
```

But this is wrong.

Given `IxStore s t a`, we can hope to get `a` only when `s=t`. If someone calls `extract` for `Store s t a`, it's his job to prove `s=t`, not ours. The unit should be equivalent to `Equal s1 s2 -> a`, not `(Equal s1 s2, a)`.

We have to write unit separately:

```
data CoIndId s s' x = CoIndId (Equal s s' -> x)
```

Now, it works well:

```
extract :: Dual IndT CoIndId m
extract :: IndT m CoIndId
extract :: forall s1 s2 a. m s1 s2 a -> CoIndId s1 s2 a
extract :: forall s1 s2 a. m s1 s2 a -> Equal s1 s2 -> a
extract :: forall s a. m s s a -> a
```

Good. Let's do `duplicate`.

```
duplicate :: Dual IndT (ICompose m m) m
duplicate :: IndT m (ICompose m m)
duplicate :: forall s1 s2 a. m s1 s2 a -> ICompose m m s1 s2 a
duplicate :: forall s1 s2 a. m s1 s2 a -> exists s. (m s1 s (m s s2 a))
```

This is wrong too. We need `forall` and not `exists`. Define

```
data CoICompose m n s1 s3 x = CoICompose (forall s2. (m s1 s2 (n s2 s3 x)))
```

and the last line changes to:

```
duplicate :: forall s1 s2 s a. m s1 s2 a -> m s1 s (m s s2 a)
```

At last we can define the indexed comonad for `IxStore`.

```
type CoIndexed = MCat (Dual IndT) CoIndId CoICompose
```

```
ixstore :: Monoid CoIndexed IxStore
```

```
ixstore = M extract duplicate
```

```
where extract :: Dual IndT CoIndId IxStore
```

```
extract = Dual (IndT (\x -> CoIndId (k x)))
```

```
duplicate :: Dual IndT (CoICompose IxStore IxStore) IxStore
```

```
duplicate = Dual (IndT (\(IxStore f x) ->
```

```
CoICompose (IxStore (IxStore f) x)))
```

```
k :: IxStore s t a -> Equal s t -> a -- Needs type signature.
```

```
k (IxStore f a) Eq = f a
```

What happens when you fix `s` and consider `IxStore s s`? This is the same as `Store s` comonad. “Endomorphisms” of an indexed comonad form a comonad.

```
fromCoIndId (CoIndId x) = x Eq
```

```
fromCoICompose (CoICompose x) = x
```

```
end'' :: Monoid CoIndexed m -> Monoid CoEndHask (m x x)
```

```
end'' (M (Dual (IndT extract)) (Dual (IndT duplicate))) =
```

```
M (Dual (NatT (Id . fromCoIndId . extract)))
```

```
(Dual (NatT (Compose . fromCoICompose . duplicate)))
```

We have now six different structures of four different kinds defined as instances of the same phenomenon.

Rings

Rings can be represented using the `Monoid` framework too, albeit clumsier than previous examples. They are monoids over abelian groups.

Consider three abelian groups G, H, I , with operation written additively. Call a function $f: G \times H \rightarrow I$ **bilinear** if it satisfies:

$$\begin{aligned} f(a + b, c) &= f(a, c) + f(b, c) \\ f(c, a + b) &= f(c, a) + f(c, b) \end{aligned}$$

For example, multiplication as a function $\mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$ is bilinear. This is the exact formulation of distributive law. More generally, in any ring R multiplication is a bilinear function.

A ring will be defined as a group R with element $1 \in R$ and a bilinear function $R \times R \rightarrow R$. This again forces us to add some constructors to get in the form `mor unit r` and `mor (prod r r) r`.

First, definition of a group and a homomorphism:

```
data Group g = Group (g -> g -> g) g (g -> g)
data GroupHom g h = GroupHom (g -> h)
```

This is the group of integers \mathbb{Z} .

```
intGroup :: Group Int
intGroup = Group (+) 0 negate
```

We assume that constructor `Group` can only be used with an abelian group, and `GroupHom` only with a group homomorphism. This cannot be enforced directly (just like it's impossible to enforce in Haskell that an instance of `Monad` type class satisfies monad laws).

Observe that when R is a group, an element $g \in R$ gives a group homomorphism $f: \mathbb{Z} \rightarrow R$ by formula $f(n) = ng$. Conversely, any homomorphism $f: \mathbb{Z} \rightarrow R$ satisfies $f(n) = nf(1)$. So the choice of the unit $1 \in R$ will be represented as `GroupHom Int r`, which fulfils the pattern for monoid identity.

```
powr :: Group g -> g -> GroupHom Int g
powr (Group m u i) g = GroupHom pow
  where pow 0 = u
        pow n | n < 0 = i (pow (-n))
        pow n = m g (pow (n-1))
```

Now, given a group R we need such notion of a product \otimes such that a homomorphism $R \otimes R \rightarrow R$ is equivalent to a bilinear function $R \times R \rightarrow R$. This product is called the **tensor product**.

A tensor product of two groups G, H is another group, whose elements are formal sums (lists) of pairs (g, h) , where we make the following identifications:

$$(a + b, c) = (a, c) + (b, c)$$

$$(c, a + b) = (c, a) + (c, b)$$

Below is the tensor product of the groups in Omega.

```
data Tensor g h = Tensor [(g,h)]

tensor :: Group g -> Group h -> Group (Tensor g h)
tensor (Group _ _ i) _ = Group (\(Tensor a) (Tensor b) -> Tensor (a ++ b))
                                (Tensor [])
                                (\(Tensor a) -> Tensor (map (first i) a))
```

Warning: Equality in `Tensor g h` is not equality of lists, but it has to account for the identities in tensor product. (This is known as quotient group.)

Theorem 1. *Every group homomorphism $f: G \otimes H \rightarrow I$ gives a bilinear function $g: G \times H \rightarrow I$, with formula $g(a, b) = f((a, b))$. Conversely, every bilinear function $g: G \times H \rightarrow I$ induces a group homomorphism $f: G \otimes H \rightarrow I$ by formula $f((a_1, b_1) + \dots + (a_n, b_n)) = g(a_1, b_1) + \dots + g(a_n, b_n)$.*

Below, the function `bilin` converts a bilinear function to a homomorphism from tensor product.

```
gsum :: Group g -> [g] -> g
gsum (Group m u i) (x:xs) = m x (gsum (Group m u i) xs)
gsum (Group _ u _) [] = u

bilin :: Group r -> (g -> h -> r) -> GroupHom (Tensor g h) r
bilin g f = GroupHom (\(Tensor xs) -> gsum g (map (\(a,b) -> f a b) xs))
```

Now, we can represent the unity $1 \in R$ as `GroupHom Int r`, and the bilinear multiplication $R \times R \rightarrow R$ as `GroupHom (Tensor r r) r`.

```
type Ab = MCat GroupHom Int Tensor

intRing :: Monoid Ab Int
intRing = M (powr intGroup 1) (bilin intGroup (*))

boolRing :: Monoid Ab Bool
boolRing = M (powr boolG True) (bilin boolG (&&))
  where boolG :: Group Bool
```

```
boolG = Group xor not False

xor False False = False
xor True True = False
xor _ _ = True
```

This type represents rings with unit, with associative but not necessarily commutative multiplication.

The missing part

I remarked on beginning that monads defined here are lacking `fmap`. Its equivalent is missing from comonads and indexed objects too. And when defining rings, we didn't really define the additive group – the definition of a ring involves defining unit and bilinear multiplication. There is a trick that allows to extract addition from this data:

```
ringadd :: Monoid Ab v -> v -> v -> v
ringadd (M (GroupHom unit) (GroupHom mult)) x y =
  mult (Tensor [(x, unit 1), (y, unit 1)])
```

but getting inverse element seems impossible, even though it is determined by addition. We are really missing the group part of a ring and the functor part of a monad.

The problem is that we defined a monoid for `c :: MCategory k` using kind `k`. However, a monad is not any monoid in `* ~> *`: it requires a functor. A ring is not any monoid in `*`: it is a monoid for an abelian group.

We need an ability to create a kind whose members are pairs (type, value). A Haskell equivalent would be a kind whose members are types that must be an instance of some type class. For example, one can imagine kind `Eq`, and a type declared, in pseudocode, like this:

```
kind Eq = (t :: *; t -> t -> Bool)

type A :: Eq
type A = (Int; equalInt)
```

ML users might recall module system, which allows to group values and types like this:

```
signature EQ =
sig
  type 'a
  val eq : 'a -> 'a -> 'a
end;;

structure A : EQ =
struct
  type 'a = int
  val eq = equalInt
end;;
```

This problem occurs in many forms; one of them is that `Set` cannot be made a monad in Haskell (because of the `Ord` constraint).

A recent extension – **constraint kinds** [5] – might be of interest.

Omega doesn't allow to group types and values as far as I know. But there is a workaround.

Meet type functions

Omega supports type functions. They are written using curly braces `{}`. Below are type functions giving components of `MCategory k`.

```
getprod :: forall (k :: *1). MCATEGORY k ~> k ~> k ~> k
{getprod (MCat mor un pro) a b} = pro a b
```

```
getunit :: forall (k :: *1). MCATEGORY k ~> k
{getunit (MCat mor un pro)} = un
```

```
getmor :: forall (k :: *1). MCATEGORY k ~> k ~> k ~> *0
{getmor (MCat mor un pro) a b} = mor a b
```

These are like `fst` and `snd`, but on the type level.

We will associate to each type of kind `MCATEGORY k` type of missing data, inherent to `k`. For example, if you want to use `Maybe :: * ~> *` as an object in `EndHask` you need a value of type `Functor Maybe`. If you want to use `Int` as an object in `Ab`, you have to supply a value of type `Group Int`.

```
data Functor f = Functor (forall x y. (x -> y) -> f x -> f y)
data IxFunctor f = IxFunctor (forall a b x y. (x -> y) -> f a b x -> f a b y)
```

```
inh :: forall (k :: *1). MCATEGORY k ~> k ~> *0
```

The Monad.Reader

```
{inh Hask m} = ()
{inh EndHask f} = Functor f
{inh CoEndHask f} = Functor f
{inh Graph g} = ()
{inh Indexed f} = IxFunctor f
{inh CoIndexed f} = IxFunctor f
{inh Ab g} = Group g
```

In general, a type `t :: k` used as an object of `c :: MCategory k` is incomplete without a value of type `{inh c t}`.

Unfortunately, type functions, like normal Haskell functions are closed. They have to be defined in one chunk of code in a single module. Like Haskell type synonyms, they have to be fully applied, and there are no higher-order type functions.

The new monoid data type now requires to give the inherent data.

```
data Monoid2 :: forall (k :: *1). MCategory k ~> k ~> *0 where
  M2 :: forall (k :: *1) (m :: k) mor unit prod.
      {inh (MCat mor unit prod) m} ->
      mor unit m -> mor (prod m m) m ->
      Monoid2 (MCat mor unit prod) m
```

We can now define proper monads with `fmap`, proper rings with addition, proper indexed monads.

```
maybeMonad2 :: Monoid2 EndHask Maybe
maybeMonad2 = M2 (FuncT fmap) (NatT (\(Id x) -> Just x)) (NatT join)
  where fmap f (Just x) = Just (f x)
        fmap _ _ = Nothing

        join (Compose (Just (Just x))) = Just x
        join _ = Nothing
```

```
intRing2 :: Monoid2 Ab Int
intRing2 = M2 intGroup (powr intGroup 1) (bilin intGroup (*))
```

```
ixstate2 :: Monoid2 Indexed IxState
ixstate2 = M2 (IxFunctor map) (IndT ret) (IndT join)
  where ret :: IndId s s' x -> IxState s s' x
        ret (IndId x) = IxState (\s -> (x,s))
        join (ICompose x) = IxState (\s -> let (y, s') = runIxState x s
                                             in runIxState y s')

        map f x = IxState (\s -> let (y, s') = runIxState x s in (f y, s'))
```

Vector spaces and algebras

If you follow David Amos's blog [6] you have heard about vector spaces and algebras. A vector space is a set of objects, called vectors, which can be added and multiplied by scalars. The type of scalars is `k`.

```
data VSpace k v = VSpace (Group v) (k -> v -> v)
```

Below I present vector space of real numbers, and of complex numbers. Since Omega doesn't have type classes, operators like `(+)` are used for `Ints`, and `(#+)` are used for `Floats`.

```
data Complex = Complex Float Float
```

```
realVS :: VSpace Float Float
realVS = VSpace (Group (#+) negateFloat 0.0) (#*)
```

```
complGroup :: Group Complex
complGroup = Group cadd czero cneg
  where cadd (Complex r1 i1) (Complex r2 i2) = Complex (r1 #+ r2) (i1 #+ i2)
        cneg (Complex r i) = Complex (negateFloat r) (negateFloat i)
        czero = Complex 0.0 0.0
```

```
complexVS :: VSpace Float Complex
complexVS = VSpace complGroup vmult
  where vmult a (Complex r i) = Complex (a #* r) (a #* i)
```

An algebra is a vector space, where vectors can be multiplied. For example, the complex numbers form an algebra over real numbers. There are other examples, like dual numbers or quaternions.

The ingredients of an algebra are: a vector space V over a field K , a unit $u \in V$ and a bilinear multiplication $m: V \times V \rightarrow V$. As in the case of rings, we can express this as linear functions $u: K \rightarrow V$ and $m: V \otimes V \rightarrow V$.

```
data Linear v w = Linear (v -> w)
type VSspaces k = MCat Linear k Tensor
{inh (VSspaces k) v} = VSpace k v
```

```
tensorSpace :: VSpace k v -> VSpace k w -> VSpace k (Tensor v w)
tensorSpace (VSpace g1 m1) (VSpace g2 m2) = VSpace (tensor g1 g2)
  (\x (Tensor a) -> Tensor (map (first (m1 x)) a))
```

```

complexAlgebra :: Monoid2 (VSpaces Float) Complex
complexAlgebra = M2 complexVS (Linear un) (Linear mult)
  where un x = Complex x 0.0

      mult :: Tensor Complex Complex -> Complex
      mult (Tensor xs) = gsum complGroup (map cmult xs)
      cmult (Complex r1 i1, Complex r2 i2) =
        Complex (r1#*r2#-i1#*i2) (r1#*i2#+r2#*i1)

```

Monoids for monoids

Time for another strange meaning of monoid.

If you have a (normal) monoid m , what is a monoid structure for m ? The unit is a monoid homomorphism from the unit monoid $()$ to m , which isn't interesting. The multiplication is a monoid homomorphism from (m, m) to m . So we have a monoid m which is equipped with another multiplication, which is a monoid homomorphism.

```

data MonoidHom m n = MonoidHom (m -> n)
type Monoids = MCat MonoidHom () (,)
{inh Monoids m} = Monoid Hask m

```

Last line has to be moved to the `inh` definition.

A monoid object m in `Monoids` is a monoid in `Hask` for m with unit `MonoidHom () m` and multiplication `MonoidHom (m,m) m`.

It turns out that monoid laws imply that such objects exist only when the two multiplications are the same and they are commutative.

See The Catsters lecture on YouTube for a video tutorial on Eckmann-Hilton argument [7] and on how a monoid object in the category of monoids is a commutative monoid [8].

Below is a monoid object for a monoid.

```

intMon :: Monoid2 Monoids Int
intMon = M2 intM (MonoidHom zero) (MonoidHom mult)
  where intM :: Monoid Hask Int
        intM = M zero mult

        mult (x,y) = x + y
        zero () = 0

```

Finishing the product

We specified previously objects like `EndHask`, giving an operation called `Compose`. `Compose` took two type constructors of kind `* ~> *` and gave `Compose f g`.

We have to refine this. `Compose` should take functors, and return a functor. In other words, it has to take `fmap` for `f` and `g`, and return `fmap` for their composition.

```
composeFunctor :: Functor f -> Functor g -> Functor (Compose f g)
composeFunctor (Functor f) (Functor g) = Functor (lift . f . g)
  where lift k (Compose x) = Compose (k x)
```

More generally, for any `c :: MCategory k` we have to specify a function of type

```
{inh c x} -> {inh c y} -> {inh c {getprod c x y}}
```

When defining this, we ran into a small issue: a user of this function doesn't identify `c`. The trick is to use a **reified type** [9] – a value whose only purpose is to identify a type. In Haskell this is used in type signature of `floatDigits` of a `RealFloat` type:

```
floatDigits :: RealFloat a => a -> Int
```

This function doesn't really use the value of its argument. It only uses its type `a`. For example, to get `floatDigits` of a `Float`, one can call `floatDigits (undefined :: Float)`. As an improvement, it's better to create a special type, which ensures the value doesn't convey any information. This is implemented in the `tagged` package on Hackage[10].

Similarly, the `unit` in `MCategory k` should bear the underlying structure, for example we need `fmap` for `data Id x = Id x`.

```
idFunctor :: Functor Id
idFunctor = Functor (\f (Id x) -> Id (f x))
```

```
idIxFunctor :: IxFunctor IndId
idIxFunctor = IxFunctor k
  where k :: (a -> b) -> IndId s s' a -> IndId s s' b -- Needed
        k f (IndId x) = IndId (f x)
```

```
idCIxFunctor :: IxFunctor CoIndId
idCIxFunctor = IxFunctor k
  where -- k :: (a -> b) -> CoIndId s s' a -> CoIndId s s' b -- Not needed
        k f (CoIndId x) = CoIndId (f . x)
```

Listing 1 defines functions `inhprod` and `inhunit`.

```

data Token :: MCategory k ~> *0 where
  Token :: forall (k :: *1) (a :: MCategory k). Token a

emptyProd () () = ()

inhprod :: forall (k :: *1) (c :: MCategory k) (x :: k) (y :: k).
  Token c -> {inh c x} -> {inh c y} -> {inh c {getprod c x y}}
inhprod (Token :: Token Hask)           = emptyProd
inhprod (Token :: Token EndHask)        = composeFunctor
inhprod (Token :: Token CoEndHask)      = composeFunctor
inhprod (Token :: Token Graph)         = emptyProd
inhprod (Token :: Token Indexed)       = composeIxFunctor
inhprod (Token :: Token CoIndexed)     = composeCIxFunctor
inhprod (Token :: Token Ab)            = tensor
inhprod (Token :: Token Monoids)       = prodmon
inhprod (Token :: Token (VSpaces k)) = tensorSpace

inhunit :: forall (k :: *1) (c :: MCategory k). Token c ->
  {inh c {getunit c}}
inhunit (Token :: Token Hask)           = ()
inhunit (Token :: Token EndHask)       = idFunctor
inhunit (Token :: Token CoEndHask)     = idFunctor
inhunit (Token :: Token Graph)         = ()
inhunit (Token :: Token Indexed)       = idIxFunctor
inhunit (Token :: Token CoIndexed)     = idCIxFunctor
inhunit (Token :: Token Ab)            = intGroup
inhunit (Token :: Token Monoids)       = M (\() -> ()) (\((),()) -> ())
inhunit (Token :: Token (VSpaces Float)) = realVS

```

Figure 1: Completing product and unit in categories.

Level polymorphism

We defined now monoids, (co)monads, categories, indexed (co)monads and other things using types of kind `MCategory k` for different `k`.

It turns out that in a way the `MCategory` kind itself is a monoid type, and `Hask`, `EndHask` and other types are particular monoids. Compare:

```
naturals :: Monoid Int           Hask :: MCategory *
0 :: Int                        () :: *
(+) :: Int -> Int -> Int       (,) :: * ~> * ~> *

endo :: Monoid (a -> a)         EndHask :: MCategory (* ~> *)
id :: a -> a                    Id :: * ~> *
(.) :: (a -> a) -> (a -> a) -> Compose :: (* ~> *) -> (* ~> *) ~>
(a -> a)                        (* ~> *)
```

However, there is no kind polymorphism here: the **levels** are wrong. Corresponding items are one level up. `naturals` is a value; `Hask` is a type. `Int` is a type; `*` is a kind.

Types of kind `MCategory k` had also `mor` type as a component. As in previous cases, `mor` is not a part of monoid: it is the data of the underlying (inherent) structure. Just like rings are abelian groups with unit of type `r` and product of type `r -> r -> r`, types of kind `MCategory k` are categories with unit of kind `k` and product of kind `k ~> k ~> k`.

Omega has support for level polymorphism. Here's a level-polymorphic version of `Monoid` and `MCategory`.

```
data MCategory' :: level n. *n ~> *n ~> *n where
  MCat' :: forall (k :: *n) (m :: *n). (k ~> k ~> m) ~>
    k ~> (k ~> k ~> k) ~> MCategory' m k

data Monoid' :: level n. forall (k :: *(n+1)) (mo :: *(n+1)).
  MCategory' mo k ~> k ~> *n where
  M' :: forall (k :: *(n+1)) (mo :: *(n+1)) (m :: k)
    (mor :: k ~> k ~> mo) (unit :: k) (prod :: k ~> k ~> k).
    mor unit m ~> mor (prod m m) m ~>
    Monoid' (MCat' mor unit prod) m
```

The declaration `level n.` signifies level polymorphism, and `n` can be later used in `*n`.

The code from previous sections works after only adding primes in several places.

The Monad.Reader

```
type Hask' = MCat' (->) () (,)
type EndHask' = MCat' NatT Id Compose

sumMonoid' :: Monoid' Hask' Int
sumMonoid' = M' (\() -> 1) (\(x,y) -> x*y)

maybeMonad' :: Monoid' EndHask' Maybe
maybeMonad' = M' (NatT (\(Id x) -> Just x)) (NatT join)
  where join (Compose (Just (Just x))) = Just x
        join _ = Nothing
```

We can now write `Hask` as a monoid, with `()` as the unit and `(,)` as multiplication.

```
kind Unit = Unt
kind Morp x y = Mrp (x ~> y)
kind Prod x y = Prd x y

type Cat = MCat' Morp Unit Prod

data UnitHsk :: Unit ~> *0 where
  UnitHskV :: UnitHsk Unt

data ProdHsk :: Prod *0 *0 ~> *0 where
  ProdHskV :: (x, y) -> ProdHsk (Prd x y)

type Hask'' = M' (Morp UnitHsk) (Morp ProdHsk) -- :: Monoid' Cat *
```

Notice `M'` was used both as a value, and as a type, and `Monoid'` is now both type of `sumMonoid'` and the kind of `Hask''`.

We can now convert a `Monoid' Cat k` into `MCategory k`:

<code>Monoid Hask m</code>	<code>haskell monoids</code>
<code>() -> m</code>	<code>m</code>
<code>(m,m) -> m</code>	<code>m -> m -> m</code>
<code>Monoid' Cat k</code>	<code>MCategory k</code>
<code>Unit ~> k</code>	<code>k</code>
<code>Prod k k ~> k</code>	<code>k ~> k ~> k</code>

We need type-level currying.

```
data Curry f x y = Curry (f (Prd x y))
```

```
crea :: Monoid' Cat *0 ~> (*0 ~> *0 ~> *0) ~> MCategory *0
{crea (M' (Mrp u) (Mrp p)) mor} = MCat mor (u Unt) (Curry p)
```

```
crea' :: Monoid' Cat *0 ~> (*0 ~> *0 ~> m) ~> MCategory' m *0
{crea' (M' (Mrp u) (Mrp p)) mor} = MCat' mor (u Unt) (Curry p)
```

Now we can use `Hask''` to build a monoid:

```
intmul' :: Monoid' {crea' Hask'' (->)} Int
intmul' = M' (\UnitHskV -> 1) (\(Curry (ProdHskV (x,y))) -> x * y)
```

Conal Elliott's blog post [11] highlighting analogies between differentiation on values and types implicitly contains level polymorphism. Dan Piponi's blog post [12] containing type-level matrices makes me think there are other instances of level polymorphism, although I don't know if they can be expressed in Omega at all.

Type constructors are not sufficient

Unfortunately, we cannot use `crea` on `EndHask'`. It takes `Monoid' Cat *0`, but not `Monoid' Cat k`. The problem is with `Curry`, whose kind is:

```
forall (a :: *1) (b :: *1). (Prod a b ~> *0) ~> a ~> b ~> *0
```

We can write manually an `Curry2` type, for different kind.

```
data Curry2 f x y z = Curry2 (f (Prd x y) z)
```

```
crea'' :: Monoid' Cat (*0 ~> *0) ~> ((*0 ~> *0) ~> (*0 ~> *0) ~> m) ~>
      MCategory' m (*0 ~> *0)
{crea'' (M' (Mrp u) (Mrp p)) mor} = MCat' mor (u Unt) (Curry2 p)
```

On the rightmost side of type constructors, there is always `*` (or `*n`). The `*n` types are always open for new members, which can be defined elsewhere. This limitation doesn't allow to write a general `Curry` type.

Similarly, the `Dual` constructor `Dual c a b = Dual (c b a)` has kind `(a ~> b ~> *) ~> b ~> a ~> *`. Notice difference with the Haskell function `flip :: (a -> b -> c) -> b -> a -> c`. For example, `Dual` works on `(->)`, `NatT` and `IndT`, but won't work on `Compose` – unlike its value-level equivalent `flip (.)`.

A simpler example is the `Id` constructor `Id x = Id x`. Its kind is `* ~> *`, unlike type of identity function `a -> a`. You can write `Id Int`, but not `Id Maybe Int`.

General view of endomorphisms

You might have noticed that some of the `MCategory k` had indexed versions, which had endomorphisms:

```
end    :: Monoid Graph m -> Monoid Hask (m x x)
end'   :: Monoid Indexed m -> Monoid EndHask (m x x)
end''  :: Monoid CoIndexed m -> Monoid CoEndHask (m x x)
```

We can write this analogy using the category datatype:

```
data Category :: forall (k :: *1). MCATEGORY k ~> (*0 ~> *0 ~> k) ~> *0 where
  C :: forall (k :: *1) (c :: *0 ~> *0 ~> k) mor unit prod.
    (forall x. mor unit (c x x)) ->
    (forall x y z. mor (prod (c x y) (c y z)) (c x z)) ->
    Category (MCat mor unit prod) c
```

As it turns out,

```
Monoid Graph c      is equivalent to Category Hask c
Monoid Indexed c    is equivalent to Category EndHask c
Monoid CoIndexed c is equivalent to Category CoEndHask c
```

In attached source code I included conversion functions going both ways.

The general endomorphism function for this type is simply

```
endGen :: Category k c -> Monoid k (c x x)
endGen (C x y) = M x y
```

This function encompasses endomorphisms of a category, indexed monad and indexed comonad.

I strongly suspect that there is a type operator `Index` such that `Graph` is essentially `Index Hask`, `Indexed` is essentially `Index EndHask` and `CoIndexed` is essentially `Index CoEndHask`, and those three functions are really one function. As far as I know, currently the type system doesn't allow to express it, for the same reason as `Curry`, `Id` etc. are not completely polymorphic.

Final remarks

We defined the following types:

```

type Hask      = MCat (->) () (,)
type EndHask   = MCat NatT Id Compose
type CoEndHask = MCat (Dual NatT) Id Compose
type Graph     = MCat GraT Equal GCompose
type Indexed   = MCat IndT IndId ICompose
type CoIndexed = MCat (Dual IndT) CoIndId CoICompose
type Ab        = MCat GroupHom Int Tensor
type Monoids   = MCat MonoidHom () (,)
type VSpaces k = MCat Linear k Tensor
type Cats      = MCat' Morp Unit Prod

```

and a monoid for each one:

```

sumMonoid      :: Monoid Hask Int
maybeMonad    :: Monoid EndHask Maybe
envComonad     :: Monoid CoEndHask (Env s)
haskCat        :: Monoid Graph (->)
ixState        :: Monoid Indexed IxState
ixStore        :: Monoid CoIndexed IxStore
intrRing       :: Monoid Ab Int
sumCMonoid     :: Monoid Monoids Int
complexAlgebra :: Monoid (VSpaces Float) Complex
EndHask''      :: Monoid' Cats (*0 ~> *0)

```

– a monoid, monad, comonad, category, indexed monad, indexed comonad, ring, commutative monoid, an algebra and a monoidal category.

Omega did the job very well. If we had an even better language, how would it help?

- ▶ Having a dependently typed language, like Agda or Coq, would allow to enforce laws that `Monoid` is really a monoid, `Functor` really a functor, `GroupHom` really a group homomorphism and so on.
- ▶ Support for kinds with (type, value) pairs instead of just types would remove the type function `inh` trick. The kind of `EndHask` is `MCategory (* ~> *)`, but it should be `MCategory Functor`. The kind of `VSpaces` is `* ~> MCategory *`, but it should be `Field ~> MCategory *`.
- ▶ With first class functions or type synonyms, we could use real identity function `Id x = x` instead of a wrapper `data Id x = Id x` and do the same in other places. That would remove a lot of clutter, for example a monad would be defined using functions `x -> m x` and `m (m x) -> m x` instead of `NatT Id m` and `NatT (Compose m m) m`. It would also remove the problem described in section “Why type constructors are not sufficient”, allowing to write general currying on types as a function.

Category theory veterans noticed I made some oversimplifications. For example, when I called the triple `mor`, `unit`, `prod` a “monoidal category” I missed most of the components of a real monoidal category – composition, identities, `prod` on morphisms, some natural transformations obeying coherence conditions. These are data which lie on a different level in Omega’s type system, and have to, at least now, be technically separated. I included them in attached source code as `morid`, `morcomp` etc.

Also there’s a difference between a weak and a strong monoidal category, and I glossed over this. You might notice unlike normal `mappend`, `(,)` is not associative, only associative up to isomorphism.

There exists a gap between “real” category theory and implementing its concepts in Haskell or Omega. I hope it will get smaller with time.

Comments on reddit [13] bring another possible use of kind polymorphism – the `Typeable` class.

Kind polymorphism is going to be added to GHC [14].

Attached `prog.hs` contains code from article. To use it, download Omega from <http://code.google.com/p/omega/downloads/list> and run `omega.exe prog.hs` (Linux or Windows).

The post that started it all is [15], which prompted an entry on my blog [16]. See Wikipedia article on monoid object [17] and Mac Lane famous book [18] for more information on monoid objects.

Loose ideas

This section contains half-baked ideas, which will be skipped or rewritten.

Todo: Think which are decent enough.

Laws

Until now, I used `type Hask = MCat (->) (,) ()` to form monoids, and remarked it’s a monoidal category. However, the precise definition is more complex. A real monoidal category has many more components: identity and composition (just like a normal category), isomorphisms for associativity `(a, (b, c)) -> ((a, b), c)`, for unit `((), a) -> a` (left and right variants).

I included a long boring list of isomorphisms in attached code.

The associativity law is about commutativity of this diagram:

$$\begin{array}{ccc} M \times (M \times M) & \dashrightarrow & (M \times M) \times M \\ | & & | \\ M \times M & & M \times M \end{array}$$

$$\backslash \quad /$$

M

After some definitions, we can write:

```

assoclaw :: Monoid2 (MCat mor unit prod) m -> (mor (prod (prod m m) m) m, mor (prod (pr
assoclaw (M2 a _ mul) = (comp mul (prod (inhprod' t a) a a a mul ident),
                    comp (comp mul (prod a a (inhprod' t a) a ident mul)) (snd (assoc t a
  where t = Token :: Token (MCat mor unit prod)
        comp = morcomp t
        prod = morprod t
        ident = morid t

```

and conditions for other diagrams:

```

lunitlaw :: Monoid2 (MCat mor unit prod) m -> (mor m m, mor m m)
lunitlaw (M2 a un mul) = (ident, comp (comp mul (prod (inhunit' t) a a a un ident)) (snd
  where t = Token :: Token (MCat mor unit prod)
        comp = morcomp t
        prod = morprod t
        ident = morid t

```

```

runitlaw :: Monoid2 (MCat mor unit prod) m -> (mor m m, mor m m)
runitlaw (M2 a un mul) = (ident, comp (comp mul (prod a a (inhunit' t) a ident un)) (snd
  where t = Token :: Token (MCat mor unit prod)
        comp = morcomp t
        prod = morprod t
        ident = morid t

```

For example, consider integers with subtraction and 0 as unit. This doesn't look like a monoid.

```

nonmonoid :: Monoid2 Hask Int
nonmonoid = M2 () (\() -> 0) (\(x,y) -> x-y)

```

```

assoctest m a b c = let (x,y) = assoclaw m
                    u = ((a,b),c)
                    in (x u, y u)

```

```

lunittest m a = let (x,y) = lunitlaw m
                in (x a, y a)

```

```

runittest m a = let (x,y) = runitlaw m
                in (x a, y a)

```

Let's check by asking Omega:

```
prompt> assoctest nonmonoid 2 3 4
```

```
(-5,3) :: (Int,Int)
```

```
prompt> lunittest nonmonoid 3
```

```
(3,-3) :: (Int,Int)
```

```
prompt> runittest nonmonoid 3
```

```
(3,3) :: (Int,Int)
```

Subtraction failed associativity test ($(x - y) - z$ is not the same as $x - (y - z)$) and left unit test (since $0 - x$ is not x). It passed right unit test, since $x - 0 = x$.

Todo: Example for a monad and nonmonad.

Type-level CPS

Perhaps the problem with type constructors can be solved by using type-level continuation passing style. Instead of `Curry f x y = Curry (f (Prod x y))` write `Curry f x y c = Curry (c (f (Prod x y)))` now `f` can be `Prd a b ~> c`.

Lax monoidal functors

Between normal categories, there are functors. Between monoidal categories, there are lax monoidal functors.

In categories section, there was a discrete category: a category with no nontrivial morphisms. Its only content was that it was a set of all Haskell objects. Just like a discrete category is a set, a discrete monoidal category is a monoid. A lax monoidal functor between such categories is a monoid homomorphism.

Applicative functor: Monoid homomorphism:

```
() -> f ()
```

```
() = f ()
```

```
(f a, f b) -> f (a, b)
```

```
mappend (f a) (f b) = f (mappend a b)
```

See chapter 7 in original Applicative paper [19] for details.

Todo: Is there a connection to "Const m" applicative functor?

The trivial monoid

```
Monoid Hask ()
Monoid EndHask Id
Monoid CoEndHask Id
Monoid Graph Equal --discrete category
```

Universal monoids

- ▶ universal monad: [Cont blog.sigfpe.com/2008/12/mother-of-all-monads.html](http://blog.sigfpe.com/2008/12/mother-of-all-monads.html)
- ▶ universal indexed monad: [IxCont blog.sigfpe.com/2009/02/beyond-monads.html](http://blog.sigfpe.com/2009/02/beyond-monads.html)
- ▶ couniversal comonad: Store
- ▶ couniversal indexed comonad?
- ▶ <http://www.haskell.org/pipermail/haskell-cafe/2010-June/079481.html>

```
univ_monad :: Monoid2 EndHask m -> m a -> Cont (m b) a
univ_monad (M2 fmap _ join) x =
  Cont (\f -> getNatT join (Compose (deFunctor fmap f x)))
```

```
univ_comonad :: Monoid2 CoEndHask w -> Store (w a) b -> w b
univ_comonad (M2 fmap _ duplicate) (Store f x) =
  deFunctor fmap f (fromCompose (getNatT (getDual duplicate) x))
```

More analogies

- ▶ Monad - writer (m,). Comonad - traced (m ->). (need monoid)
- ▶ Monad - reader (r ->). Comonad - coreader (r,) (environment)
- ▶ Monad - state. Comonad - store?
- ▶ Monad - identity. Comonad - identity.

```
fold :: (f a -> a) -> Y f -> a == Y f -> Cont (f a) a
unfold :: Store (f a) a -> Y f
```

```
since
Y f = Y (f (Y f))
perhaps an indexed version has sense?
Y f x y = Y (f x z (Y f z y))
```

Think about

1. Work on Omega issue 99 so that level polymorphism will not need Morp.
2. Think more about Typeable

3. Think more about constraint kinds
4. Think more about Agda or Coq. Is that universe polymorphism?
5. Some propaganda on product kinds. When you have product kinds, multi-parameter type classes are normal classes. If you pack all members of a type class to a tuple, you'll get a function from types to values.
6. Think/ask if indexed comonads appear anywhere

Free monoids

Below are some code fragments in GHC you might compare.

- Free monoid (also known as lists):

```
data Free a = Nil | Cons a (Free a)

instance Monoid (Free a) where
  mempty = Nil
  mappend Nil x = x
  mappend (Cons x xs) y = Cons x (mappend xs y)
```

- Free monad (this code is copied from Haskell wiki [20])

```
data FreeMonad f = Return a | Roll (f (Free f a))

instance Monad (FreeMonad f) where
  return = Return
  join (Return fa) = fa
  join (Roll ffa) = Roll (fmap join ffa)
```

- Free category (paths):

```
data FreeCat g a b where
  NilC :: FreeCat g a a
  ConsC :: g b c -> FreeCat g a b -> FreeCat g a c

instance Category (FreeCat g) where
  id = NilC
  NilC . x = x
  ConsC x xs . y = ConsC x (xs . y)
```

- Free indexed monad:

```
data Equal a b where
  Eq :: Equal a a

data Free f x y a = Return (Equal x y) a | forall z. Roll (f x z (Free f
```

```
instance IxFunctor f => IxFunctor (Free f) where
  imap f (Return Eq x) = Return Eq (f x)
  imap f (Roll a) = Roll (imap (imap f) a)

instance IxFunctor f => IxMonad (Free f) where
  ireturn = Return Eq
  ijoin (Return Eq x) = x
  ijoin (Roll x) = Roll (imap join x)
```

- The tensor algebra [21]
- Unifying them requires adding coproduct.

References

- [1] Brent Yorgey. The Typeclassopedia. <http://www.haskell.org/wikiupload/8/85/TMR-Issue13.pdf>.
- [2] <http://www.cs.uu.nl/wiki/UHC>.
- [3] <http://code.google.com/p/omega>.
- [4] Dan Piponi. Beyond monads. <http://blog.sigfpe.com/2009/02/beyond-monads.html>.
- [5] <http://blog.omega-prime.co.uk/?p=127>.
- [6] <http://haskellformaths.blogspot.com>.
- [7] Eckmann-Hilton 1. <http://www.youtube.com/watch?v=Rjdo-RWQVIY>.
- [8] Monoid objects 2. <http://www.youtube.com/watch?v=7Sf3Y4sesZE>.
- [9] Traits type class. http://www.haskell.org/haskellwiki/Traits_type_class.
- [10] <http://hackage.haskell.org/package/tagged>.
- [11] Conal Elliott. Differentiation of higher-order types. <http://conal.net/blog/posts/differentiation-of-higher-order-types>.
- [12] Dan Piponi. Constraining Types with Regular Expressions. <http://blog.sigfpe.com/2010/08/constraining-types-with-regular.html>.
- [13] Peaker and doliorules. reddit comments. http://www.reddit.com/r/haskell/comments/cudwg/using_kind_polymorphism_monads_as_monoids/c0ve85r.
- [14] <http://www.haskell.org/pipermail/cvs-ghc/2011-September/065590.html>.

- [15] Dan Piponi. From Monoids to Monads. <http://blog.sigfpe.com/2008/11/from-monoids-to-monads.html>.
- [16] Krzysztof Gogolewski. Kind polymorphism in action. <http://monoidal.blogspot.com/2010/07/kind-polymorphism-in-action.html>.
- [17] Monoid (category theory) – Wikipedia. [http://en.wikipedia.org/w/index.php?title=Monoid_\(category_theory\)](http://en.wikipedia.org/w/index.php?title=Monoid_(category_theory)).
- [18] Saunders Mac Lane. Categories for the working mathematician (1998).
- [19] Conor McBride and Ross Paterson. Applicative programming with effects. <http://www.soi.city.ac.uk/~ross/papers/Applicative.html>.
- [20] http://www.haskell.org/haskellwiki/Free_structure.
- [21] <http://haskellformaths.blogspot.com/2011/07/tensor-algebra-monad.html>.