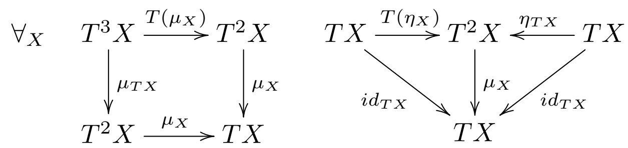
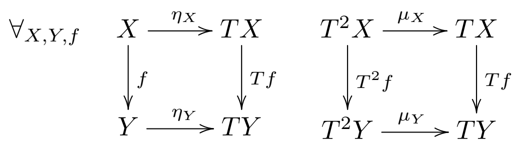
1 Expanding The Diagrams

The definition and diagrams given in ACC (§20.1) are sort of terse, so I have taken the liberty of applying the notation in (§6.2) and (§3.23, footnote 15) and applying the functors to particular objects $X, Y \in \mathbf{X}$:

- The definition now reads as: A **monad** on $\mathbf{X}$ is $(T : \mathbf{X} \to \mathbf{X}, \eta : id_{\mathbf{X}} \dot{\to} T, \mu : T^2 \dot{\to} T)$ s.t.



- The naturality conditions unpack to be



2 Translation into Haskell

`http://en.wikibooks.org/wiki/Haskell/Category_theory#The_monad_laws_and_their_importance` may be of use, and I am going to try a brief, more equational, exposition here (with many more parens than strictly necessary; deal with it).

η pretty clearly corresponds to `return`, and Tf is `fmap f`. The naturality condition on η is clear:

```
    fmap f . return   -- top right
==  return . f        -- bottom left
```

This is properly read as a constraint (part of the definition) of `return` (η) in terms of `fmap` applied at the type (constructor / functor) associated with our monad (i.e., the morphism part of the functor).

Similarly, μ corresponds to `join`, whose Haskell definition is

```
join :: m (m a) -> m a
join mma = (mma >>= id)    -- or just "join = (>>= id)"
```

(Haskell, by convention, uses `m` for T; sorry for the confusion.) Its naturality condition says that

```
    (fmap f) . join          -- top right
==  join . (fmap (fmap f))   -- bottom left
```

This says, basically, that you can first run your inner monadic thingie and then apply a "lifted" function to the result, or you can lift the function twice, so that it applies inside your inner monadic thingie and *then* run the inner thing. Again, this should be taken as a constraint (part of the definition) on join (μ) in terms of `fmap`.

And now the other two laws, which are more interesting and can nicely be executed in terms of Haskell's `>>=`. First, we have:

$$\mu_X \circ T(\mu_X) = \mu_X \circ \mu_{TX}$$

or

```
join . (fmap join) == join . join
```

Which is easy enough to see:

```
  join (fmap join mmmx)
= (mmmx >>= return . join) >>= id                     -- defn join, fmap
= (mmmx >>= (\mmx -> return (join mmx))) >>= id       -- syntax
= (mmmx >>= (\mmx -> return (mmx >>= id))) >>= id     -- defn join
= mmmx >>= (\mmx -> (return (mmx >>= id) >>= id))     -- assoc >>=
= mmmx >>= (\mmx -> id (mmx >>= id))                  -- left-identity >>=
= mmmx >>= (\mmx -> mmx >>= id)                       -- apply
= mmmx >>= (\mmx -> mmx) >>= id                       -- assoc >>=
= (mmmx >>= id) >>= id                                -- assoc >>=
= join (join mmmx)                                    -- defn join, join
```

If we label our mmmx object as $\texttt{m}_1\texttt{m}_2\texttt{m}_3\texttt{x}$, this says, in some pseudo-notation, that $\texttt{join}(\texttt{m}_1(\texttt{m}_{23}\texttt{x}))$ is the same as $\texttt{join}(\texttt{m}_{12}(\texttt{m}_3\texttt{x}))$.

And for the last, we have:

$$id = \mu_X \circ T(\eta_X) = \mu_X \circ \eta_{TX}$$

or

```
id == join . (fmap return) == \tx -> join . (return tx)
```

(note that we do not interpret η_{TX} as `return . return`, but as `return tx`! There's no guarantee that a (generalized) element of TX is the result of η_X – consider, for example, the `Either e` (i.e., $(e+)$) monads!) which again admits executable rewriting (I have taken the liberty of subscripting some functions, just to make the rewrites clearer.)

```
  join (fmap return tx)
= join (tx >>= return_1 . return_2)                 -- defn fmap
= (tx >>= return_1 . return_2) >>= id               -- defn join
= (tx >>= (\x -> return_1 (return_2 x))) >>= id    -- syntax
= tx >>= (\x -> ((return_1 (return_2 x)) >>= id)   -- assoc >>=
= tx >>= (\x -> id (return_2 x))                    -- left-identity >>=
= tx >>= (\x -> return_2 x)                         -- apply
= tx >>= return_2                                   -- syntax
= tx                                                -- right-identity >>=
```

and

```
  \tx -> join . (return tx)
= \tx -> ((return tx) >>= id)                       -- defn join
= \tx -> (id tx)                                    -- left-identity >>=
= \tx -> tx                                         -- apply
= id                                                -- defn
```

3 Speaking of Haskell

For the curious, `>>=` can be implemented in terms of join (which makes the above arguments circular (sorry!), but puts Haskell's typical treatment of Monads on firmer ground):

```
(>>=) :: m a -> (a -> m b) -> m b
ma >>= f  = join (fmap f ma)
--        = (fmap f ma) >>= id
--        = (ma >>= return . f) >>= id
--        = ma >>= (\a -> (return (f a) >>= id))
--        = ma >>= (\a -> id (f a))
--        = ma >>= \a -> f a
--        = ma >>= f
```