About the `Applicative` class

Xun Zhu

`<*>` solves the composibility issue of `fmap`

`fmap` has the following signature:

```
fmap :: (a -> b) -> f a -> f b
```

It "lifts" a function into the functor space, like so:

[center] | g :: a -> b | fmap g :: f a -> f b

There seems to have no problem until you want to lift a function that takes in more than one parameter. Since we are working with Haskell, by "a function that takes in more than one parameter" I meant *a higher order function.* That is, we want to apply `fmap` in some manner (and possibly more than once) to lift

```
g :: a -> b -> c
```

into

```
liftedG :: f a -> f b -> f c
```

Let's see what happens if we apply `fmap` once:

```
     g :: a -> b -> c
fmap g :: f a -> f (b -> c)
```

Hum … if only we could lift the `b -> c` in the `f` functor into `f b -> f c` we would be done. Unfortunately, with only `fmap` we cannot do it. Now, introduce `<*>`:

```
(<*>) :: f (b -> c) -> f b -> f c
```

This function solves exactly the problem we faced:

```
        fmap g :: f a -> f (b -> c)
(<*>) . fmap g :: f a -> f b -> f c
```

In fact, with a trivial helper function that lifts any element into the functor space:

```
pure :: a -> f a
```

from which `fmap` can be defined as:

```
    fmap = (<*>) . pure
  fmap g = (<*>) (pure g)
fmap g x = (<*>) (pure g) x
fmap g x = pure g <*> x
```

we don't even need the `fmap` anymore:

```
                (<*>) . fmap g :: f a -> f b -> f c
        \x -> (<*>) (fmap g x) :: f a -> f b -> f c
    \x -> (<*>) (pure g <*> x) :: f a -> f b -> f c
\x y -> (<*>) (pure g <*> x) y :: f a -> f b -> f c
  \x y -> (pure g <*> x) <*> y :: f a -> f b -> f c
    \x y -> pure g <*> x <*> y :: f a -> f b -> f c
```

Now it's easy to see that this new `<*>` scheme can be generated to any number of parameters:

```
                                 g :: a -> b -> c -> d
\x y z -> pure g <*> x <*> y <*> z :: f a -> f b -> f c -> f d
```

`sequenceA` does not preserve length.

If

`x, y, z :: Applicative f => f a`

then it is always the case that

`w = sequenceA [x, y, z] :: Applicative f => f [a]`

However `w` does not necessarily have length 3. It does when `f = IO`.