Functional Programming for Logicians

Péter Mekis

Department of Logic, ELTE Budapest

Session 9: 2019 April 15

Functor laws

Definition

```
class Functor f where
  fmap :: (a -> b) ->f a -> f b
```

Functor laws

Definition

```
class Functor f where
  fmap :: (a -> b) ->f a -> f b
```

Identity

```
fmap id == id
```

Functor laws

Definition

```
class Functor f where
  fmap :: (a -> b) ->f a -> f b
```

Identity

```
fmap id == id
```

Composition

```
fmap (f . g) == fmap f . fmap . g
```

Applicative laws

Definition

```
class (Functor f) =>  Applicative f where
  pure :: a -> f a
  <*> :: f (a -> b) -> f a -> f b
```

Applicative laws

Definition

```
class (Functor f) =>  Applicative f where
  pure :: a -> f a
  <*> :: f (a -> b) -> f a -> f b
```

Identity

```
pure id <*> xc == xc
```

Applicative laws

Definition

```
class (Functor f) =>  Applicative f where
  pure :: a -> f a
  <*> :: f (a -> b) -> f a -> f b
```

Identity

```
pure id <*> xc == xc
```

Homomorphism

```
pure f <*> pure x == pure (f x)
```

Applicative laws

Definition

```
class (Functor f) =>  Applicative f where
  pure :: a -> f a
  <*> :: f (a -> b) -> f a -> f b
```

Identity

```
pure id <*> xc == xc
```

Homomorphism

```
pure f <*> pure x == pure (f x)
```

Interchange

```
fc <*> pure x == pure ($ x) <*> fc
```

Applicative laws

Definition

```
class (Functor f) =>  Applicative f where
  pure :: a -> f a
  <*> :: f (a -> b) -> f a -> f b
```

Identity

```
pure id <*> xc == xc
```

Homomorphism

```
pure f <*> pure x == pure (f x)
```

Interchange

```
fc <*> pure x == pure ($ x) <*> fc
```

Composition

```
pure (.) <*> fc <*> gc <*> hc == fc <*> (gc <*> hc)
```

Monad laws

Definition

```
class (Applicative m) =>  Monad m where
  return :: a -> m a
  >>= :: m a -> (a -> m b) -> m b
```

Monad laws

Definition

```
class (Applicative m) =>  Monad m where
  return :: a -> m a
  >>= :: m a -> (a -> m b) -> m b
```

Left identity

```
return x >>= fl == fl x
```

Monad laws

Definition

```
class (Applicative m) =>  Monad m where
  return :: a -> m a
  >>= :: m a -> (a -> m b) -> m b
```

Left identity

```
return x >>= fl == fl x
```

Right identity

```
xc >>= return == xc
```

Monad laws

Definition

```
class (Applicative m) =>  Monad m where
  return :: a -> m a
  >>= :: m a -> (a -> m b) -> m b
```

Left identity

```
return x >>= fl == fl x
```

Right identity

```
xc >>= return == xc
```

Associativity

```
xc >>= (\y -> fl y >>= gl) == (xc >>= fl) >>= gl
```

Monad laws

Definition

```
class (Applicative m) =>  Monad m where
  return :: a -> m a
  >>= :: m a -> (a -> m b) -> m b
```

Left identity

```
return x >>= fl == fl x
```

Right identity

```
 xc >>= return == xc
```

Associativity

```
xc >>= (\y -> fl y >>= gl) == (xc >>= fl) >>= gl
```

Compatibility

```
return == pure
```

Fancy infix symbols

- $ Apply

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

Fancy infix symbols

`$` Apply

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

`<$>` Map

```
(<$>) :: Functor f => (a -> b) ->f a -> f b
f <$> xc = fmap f xc
```

Fancy infix symbols

- `$` Apply
  ```
  ($) :: (a -> b) -> a -> b
  f $ x = f x
  ```
- `<$>` Map
  ```
  (<$>) :: Functor f => (a -> b) ->f a -> f b
  f <$> xc = fmap f xc
  ```
- `<*>` App
  ```
  (<*>) :: Applicative f => f (a -> b) -> f a -> f b
  (<*>) fc xc == app fc xc
  ```

Fancy infix symbols

`$` Apply

```haskell
($) :: (a -> b) -> a -> b
f $ x = f x
```

`<$>` Map

```haskell
(<$>) :: Functor f => (a -> b) ->f a -> f b
f <$> xc = fmap f xc
```

`<*>` App

```haskell
(<*>) :: Applicative f => f (a -> b) -> f a -> f b
(<*>) fc xc == app fc xc
```

`>>=` Bind

```haskell
(>>=) :: Monad m => m a -> (a -> m b) -> m b
(>>=) xc fl == bind xc fl
```

Fancy infix symbols

$ Apply

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

<$> Map

```
(<$>) :: Functor f => (a -> b) ->f a -> f b
f <$> xc = fmap f xc
```

<*> App

```
(<*>) :: Applicative f => f (a -> b) -> f a -> f b
(<*>) fc xc == app fc xc
```

>>= Bind

```
(>>=) :: Monad m => m a -> (a -> m b) -> m b
(>>=) xc fl == bind xc fl
```

>> Sequence

```
(>>) :: Monad m => m a -> m b -> m b
(>>=) xc yc == xc >>= (\z -> yc)
```