Haskell and Domain-Specific Languages

Haskell nejen pro informatiky

Otakar Smrž

Institute of Formal and Applied Linguistics
Faculty of Mathematics and Physics
Charles University in Prague
otakar.smrz@mff.cuni.cz

https://wiki.ufal.ms.mff.cuni.cz/courses:pfl080
Part I

Monad Laws
class Functor \( f \) where

\[
\text{fmap} :: (a \rightarrow b) \rightarrow (f a \rightarrow f b)
\]
Functor Laws

class Functor f where

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

fmap id ≡ id
class Functor \( f \) where

\[
\text{fmap} :: (a \rightarrow b) \rightarrow (f a \rightarrow f b)
\]

\[
\text{fmap \ id} \equiv \ id
\]

\[
\text{fmap \ (f \ . \ g)} \equiv \text{fmap \ f} \ . \ \text{fmap \ g}
\]
class Functor \( f \) where

\[
fmap :: (a \rightarrow b) \rightarrow (f a \rightarrow f b)
\]

\[
fmap \ id \equiv id
\]

\[
fmap (f \ . \ g) \equiv fmap f \ . \ fmap g
\]

\[
fmap f \ xs \equiv xs >>= return \ . \ f
\]
class Functor f where
    fmap :: (a -> b) -> (f a -> f b)

fmap id ≡ id
fmap (f . g) ≡ fmap f . fmap g
fmap f xs ≡ xs >>= return . f

instance Functor [] where    fmap = map

instance Functor Tree where
    fmap f (Node a t) = Node (f a) (map (fmap f) t)
 Monad Laws

class Monad m where

(>>=) :: m a -> (a -> m b) -> m b
(>>) :: m a -> m b -> m b
return :: a -> m a
fail :: String -> m a
class Monad m where

  (>>=)   :: m a -> (a -> m b) -> m b
  (>>)    :: m a -> m b -> m b
  return  :: a -> m a
  fail    :: String -> m a

return a >>= k ≡ k a
class Monad m where

  (>>=) :: m a -> (a -> m b) -> m b
  (>>) :: m a -> m b -> m b
  return :: a -> m a
  fail :: String -> m a

return a >>= k ≡ k a
m >>= return ≡ m
class Monad m where

  (>>=) :: m a -> (a -> m b) -> m b

  (>>) :: m a -> m b -> m b

  return :: a -> m a

  fail :: String -> m a

return a >>= k ≡ k a

m >>= return ≡ m

m >>= (\ x -> h x >>= g) ≡ m >>= h >>= g
Monad Laws

```haskell
class Monad m where

  (>>=) :: m a -> (a -> m b) -> m b

  (>>) :: m a -> m b -> m b

  return :: a -> m a

  fail :: String -> m a

return a >>= k  ≡  k a

m >>= return  ≡  m

m >>= (\ x -> h x >>= g)  ≡  (m >>= h) >>= g
```

```haskell
instance Monad [] where (>>=) x f = concat (map f x)
          return x = [x]
          fail     = []
```
Monad Laws

class Monad m where

(>>=) :: m a -> (a -> m b) -> m b
(>>) :: m a -> m b -> m b
return :: a -> m a
fail :: String -> m a

return a >>= k ≡ k a
m >>= return ≡ m
m >>= (\ x -> h x >>= g) ≡ (m >>= \ x -> h x) >>= g

instance Monad [] where

(>>=) x f = concat (map f x)
return x = [x]
fail = []
class Monad m => MonadPlus m where

  mzero :: m a

  mplus :: m a -> m a -> m a

The IO monad is not an instance of the MonadPlus class, since it has no mzero that would satisfy the m >>= const mzero law \[1\].
class Monad m => MonadPlus m where

  mzero :: m a

  mplus :: m a -> m a -> m a

m 'mplus' mzero  ≡  m
MonadPlus Laws

class Monad m => MonadPlus m where

  mzero :: m a

  mplus :: m a -> m a -> m a

m 'mplus' mzero ≡ m

mzero 'mplus' m ≡ m

The IO monad is not an instance of the MonadPlus class, since it has no mzero that would satisfy the m >>= const mzero law [1].
class Monad m => MonadPlus m where

  mzero :: m a
  mplus :: m a -> m a -> m a

m `mplus` mzero ≡ m
m m >>= \_ -> mzero ≡ mzero
mzero `mplus` m ≡ m
class Monad m => MonadPlus m where

    mzero :: m a
    mplus :: m a -> m a -> m a

m `mplus` mzero ≡ m
m >>= const mzero ≡ mzero
mzero `mplus` m ≡ m
mzero >>= k ≡ mzero

The IO monad is not an instance of the MonadPlus class, since it has no mzero that would satisfy the m >>= const mzero law [1].
MonadPlus Laws

class Monad m => MonadPlus m where

    mzero :: m a

    mplus :: m a -> m a -> m a

m `mplus` mzero ≡ m
m >>\= const mzero ≡ mzero
mzero `mplus` m ≡ m
mzero >>\= k ≡ mzero

instance MonadPlus [] where

    mzero = []

    mplus = (++)
MonadPlus Laws

class Monad m => MonadPlus m where

  mzero :: m a
  mplus :: m a -> m a -> m a

  m `mplus` mzero ≡ m
  m `mplus` m ≡ m `mplus` mzero

instance MonadPlus [] where mzero = []
  mplus = (++)

The IO monad is not an instance of the MonadPlus class, since it has no mzero that would satisfy the m `mplus` const mzero law (1).
Exercises

Formulate the monad laws using the do notation. Discuss any analogies with the notations used in imperative languages.
Exercises

Formulate the monad laws using the do notation. Discuss any analogies with the notations used in imperative languages.

Relate function application \( f \ (g \ x) \) to monadic binding \( g \ x >>= f \) using for instance the Maybe instance (1).
Formulate the monad laws using the do notation. Discuss any analogies with the notations used in imperative languages.

Relate function application $f \ (g \ x)$ to monadic binding $g \ x >>= f$ using for instance the Maybe instance (1).

```haskell
instance Monad Maybe where
    Just x >>= k = k x
    Nothing >>= k = Nothing
    return = Just
    fail s = Nothing

instance MonadPlus Maybe where
    mzero = Nothing
    Nothing `mplus` ys = ys
    xs `mplus` ys = xs
```
Formulate the monad laws using the do notation. Discuss any analogies with the notations used in imperative languages.

Relate function application $f \; \&\& \; g \; x$ to monadic binding $f \; =<< \; g \; x$ using for instance the Maybe instance (1).

instance Monad Maybe where
  Just x >>= k = k x
  Nothing >>= k = Nothing
  return = Just
  fail s = Nothing

instance MonadPlus Maybe where
  mzero = Nothing
  Nothing `mplus` ys = ys
  xs `mplus` ys = xs
Part II

Input/Output
module Main where

main :: IO ()
main = return ()

Explore relevant modules, e.g. System.IO, System.Environment, and note the command :-main in Hugs.
module Main where

main :: IO ()
main = return ()

interact :: (String -> String) -> IO ()
interact f = getContents >>= putStrLn . f
module Main where

main :: IO ()
main = return ()

interact :: (String -> String) -> IO ()
interact f = getContents >>= putStrLn . f

print :: Show a => a -> IO ()
print = putStrLn . show
module Main where

main :: IO ()
main = return ()

interact :: (String -> String) -> IO ()
interact f = getContents >>= putStrLn . f

print :: Show a => a -> IO ()
print = putStrLn . show

Paul Hudak.

*The Haskell School of Expression: Learning Functional Programming through Multimedia.*