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

Functor Laws

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

Functor Laws

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

```
fmap id ≡ id
```

Functor Laws

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

```haskell
fmap id  ≡  id
fmap (f . g)  ≡  fmap f . fmap g
```

Functor Laws

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

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

Functor Laws

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

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

```haskell
instance Functor [] where    fmap = map
```

```haskell
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
```

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
```

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
```

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
```

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

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 >>= h) >>= g
```

```
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         = []
```

MonadPlus Laws

```
class Monad m => MonadPlus m where
    mzero :: m a
    mplus :: m a -> m a -> m a
```

MonadPlus Laws

```
class Monad m => MonadPlus m where
    mzero :: m a
    mplus :: m a -> m a -> m a
```

m `mplus` mzero ≡ m

MonadPlus Laws

```haskell
class Monad m => MonadPlus m where
    mzero :: m a
    mplus :: m a -> m a -> m a
```

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

MonadPlus Laws

```
class Monad m => MonadPlus m where
    mzero :: m a
    mplus :: m a -> m a -> m a
```

```
m `mplus` mzero ≡ m        m >>= \ _ -> mzero ≡ mzero
mzero `mplus` m ≡ m
```

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

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

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).

```
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
```

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 `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

The Pure World

```haskell
module Main where

main :: IO ()
main  = return ()
```

The Pure World

```haskell
module Main where

main :: IO ()
main  = return ()

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

The Pure World

```
module Main where

main :: IO ()
main  = return ()

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

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

The Pure World

```
module Main where

main :: IO ()
main  = return ()

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

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

Explore relevant modules, e.g. `System.IO`, `System.Environment`, `System.Console.GetOpt`, and note the command `:main` in Hugs.

References

Paul Hudak.
The Haskell School of Expression: Learning Functional Programming through Multimedia.
Cambridge University Press, 2000.