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

Monads

Kinds of Types

Type constructors can have different kinds (3, 1): `*`, `* -> *`, etc.

Kinds of Types

Type constructors can have different kinds (3, 1): `*`, `* -> *`, etc.

`* -> * -> *` $\equiv$ `* -> (* -> *)` but not `(* -> *) -> *`

Kinds of Types

Type constructors can have different kinds (3, 1): `*`, `* -> *`, etc.

`* -> * -> *` $\equiv$ `* -> (* -> *)` but not `(* -> *) -> *`

$order(*) = 0$ $\quad order(\kappa \texttt{ -> } \nu) = max(order(\kappa) + 1, order(\nu))$

Kinds of Types

Type constructors can have different kinds (3, 1): `*`, `* -> *`, etc.

`* -> * -> *` $\equiv$ `* -> (* -> *)` but not `(* -> *) -> *`

$order(*) = 0$ $\quad order(\kappa \texttt{ -> } \nu) = max(order(\kappa) + 1, order(\nu))$

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

instance Functor [] where    fmap = map
```

Kinds of Types

Type constructors can have different kinds (3, 1): `*`, `* -> *`, etc.

`* -> * -> *` $\equiv$ `* -> (* -> *)` but not `(* -> *) -> *`

$order(*) = 0$ $\quad order(\kappa \texttt{ -> } \nu) = max(order(\kappa) + 1, order(\nu))$

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

instance Functor [] where    fmap = map

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

Kinds of Types

Type constructors can have different kinds (3, 1): `*`, `* -> *`, etc.

`* -> * -> *` $\equiv$ `* -> (* -> *)` but not `(* -> *) -> *`

$order(*) = 0$ $\quad order(\kappa \texttt{ -> } \nu) = max(order(\kappa) + 1, order(\nu))$

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

instance Functor [] where    fmap = map

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

instance Functor (BinTree a) where
    fmap f (Fork a l r) = Fork a (fmap f l) (fmap f l)
    fmap f (Leaf b) = Leaf (f b)
```

Monads

Monads are `* -> *` types 'providing' values of some other type but 'encapsulating' more information, with methods and 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

    x >> y      = x >>= \ _ -> y
    fail s      = error s

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

```
instance Monad [] where (>>=) x f      = concat (map f x)

                        return x       = [x]
                        fail           = []
```

```
instance Monad [] where (x:xs) >>= f  = f x ++ (xs >>= f)
                        []     >>= f  = []
                        return x      = [x]
                        fail s        = []

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

```
instance Monad [] where (x:xs) >>= f   = f x ++ (xs >>= f)
                        []     >>= f   = []
                        return x       = [x]
                        fail s         = []

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

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

```
instance Monad [] where (x:xs) >>= f  = f x ++ (xs >>= f)
                        []     >>= f  = []
                        return x      = [x]
                        fail s        = []

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

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
```

“Do” Syntax

Easy notations to use with monadic types, directly translated to `(>>=)`, **`return`** and the like, and to λ-expressions.

```
[ (x, y) | x <- [1 .. 3], y <- [5, 3 .. -3], x /= y ]
```

“Do” Syntax

Easy notations to use with monadic types, directly translated to `(>>=)`, **`return`** and the like, and to λ-expressions.

```
[ (x, y) | x <- [1 .. 3], y <- [5, 3 .. -3], x /= y ]
```

$\Longrightarrow$

```
[(1,5),(1,3),       (1,-1),(1,-3),
 (2,5),(2,3),(2,1),(2,-1),(2,-3),
 (3,5),      (3,1),(3,-1),(3,-3)]
```

“Do” Syntax

Easy notations to use with monadic types, directly translated to `(>>=)`, **`return`** and the like, and to λ-expressions.

```
[ (x, y) | x <- [1 .. 3], y <- [5, 3 .. -3], x /= y ]
```

$\Longrightarrow$

```
[(1,5),(1,3),       (1,-1),(1,-3),
 (2,5),(2,3),(2,1),(2,-1),(2,-3),
 (3,5),      (3,1),(3,-1),(3,-3)]
```

```
do x <- [1 .. 3]
   y <- [5, 3 .. -3]
   True <- return (x /= y)
   return (x, y)
```

"Do" Syntax

Easy notations to use with monadic types, directly translated to `(>>=)`, **`return`** and the like, and to λ-expressions.

```
[ (x, y) | x <- [1 .. 3], y <- [5, 3 .. -3], x /= y ]
```

$\Longrightarrow$

```
[(1,5),(1,3),       (1,-1),(1,-3),
 (2,5),(2,3),(2,1),(2,-1),(2,-3),
 (3,5),      (3,1),(3,-1),(3,-3)]
```

```
do x <- [1 .. 3]
   y <- [5, 3 .. -3]
   True <- return (x /= y)
   return (x, y)
```

```
[1 .. 3]       >>= \ x ->
[5, 3 .. -3]   >>= \ y ->
return (x /= y) >>= \ r ->
case r of
    True -> return (x, y)
    _    -> fail "i.e. []"
```

Part II

Parsing

Showing

```
class Show a where
    show              :: a -> String
    showsPrec         :: Int -> a -> String -> String

    show x             = showsPrec 0 x ""
    showsPrec _ x s    = show x ++ s
```

Showing

```
class Show a where
    show              :: a -> String
    showsPrec         :: Int -> a -> String -> String

    show x             = showsPrec 0 x ""
    showsPrec _ x s    = show x ++ s

data BTree a = BFork (BTree a) (BTree a) | BLeaf a

instance Show a => Show (BTree a) where

    show (BLeaf x)      = show x
    show (BFork l r)    = "<" ++ show l ++ "|"
                              ++ show r ++ ">"
```

Reading

```
class Read a where
    readsPrec :: Int -> String -> [(a, String)]

reads    :: Read a => String -> [(a, String)]
reads     = readsPrec 0
```

Reading

```
class Read a where
    readsPrec :: Int -> String -> [(a, String)]

reads    :: Read a => String -> [(a, String)]
reads     = readsPrec 0

read     :: Read a => String -> a
read s    = case [ x | (x, t) <- reads s,
                       ("", "") <- lex t ] of
              [x] -> x
              []  -> error "Prelude.read: no parse"
              _   -> error "Prelude.read: ambiguous parse"
```

Reading

```
class Read a where
    readsPrec :: Int -> String -> [(a, String)]

reads    :: Read a => String -> [(a, String)]
reads     = readsPrec 0

read     :: Read a => String -> a
read s    = case [ x | (x, t) <- reads s,
                       ("", "") <- lex t ] of
              [x] -> x
              []  -> error "Prelude.read: no parse"
              _   -> error "Prelude.read: ambiguous parse"
```

Just like we have generalized showing to pretty-printing, we will see reading as a case of parsing indeed.

```
instance Read a => Read (BTree a) where

    readsPrec _   = readsBTree
```

```
instance Read a => Read (BTree a) where

    readsPrec _   = readsBTree


readsBTree    :: Read a => String -> [(BTree a, String)]

readsBTree ('<':s) = [ (BFork l r, u) |
                                (l, '|':t) <- readsBTree s,

                                (r, '>':u) <- readsBTree t ]


readsBTree s       = [ (BLeaf x, t)   | (x, t) <- reads s ]
```

```
instance Read a => Read (BTree a) where

    readsPrec _   = readsBTree


readsBTree    :: Read a => String -> [(BTree a, String)]

readsBTree s = [ (BFork l r, x) | ("<", t) <- lex s,
                                  (l, u) <- readsBTree t,
                                  ("/", v) <- lex u,
                                  (r, w) <- readsBTree v,
                                  (">", x) <- lex w ]

             ++ [ (BLeaf x, t)    | (x, t) <- reads s ]
```

The Parser

Review the functional pearl paper by Hutton and Meijer (2).

```
type Parser a = String -> [(a, String)]
```

The Parser

Review the functional pearl paper by Hutton and Meijer (2).

```
data Parser a = Parser (String -> [(a, String)])
```

The Parser

Review the functional pearl paper by Hutton and Meijer (2).

```
newtype Parser a = Parser (String -> [(a, String)])
```

The Parser

Review the functional pearl paper by Hutton and Meijer (2).

```
newtype Parser a = Parser (String -> [(a, String)])

parse               :: Parser a -> String -> [(a, String)]
parse (Parser p) s   = p s
```

The Parser

Review the functional pearl paper by Hutton and Meijer (2).

```
newtype Parser a = Parser (String -> [(a, String)])

parse               :: Parser a -> String -> [(a, String)]
parse (Parser p) s   = p s

itemParser    :: Parser Char
itemParser     = Parser ( \ s -> case s of c:r -> [(c, r)]
                                           ""  -> [] )
```

The Parser

Review the functional pearl paper by Hutton and Meijer (2).

```
newtype Parser a = Parser (String -> [(a, String)])

parse               :: Parser a -> String -> [(a, String)]
parse (Parser p) s   = p s

itemParser    :: Parser Char
itemParser     = Parser ( \ s -> case s of c:r -> [(c, r)]
                                           ""  -> [] )

consParser    :: a -> Parser a
consParser x   = Parser ( \ s -> [(x, s)] )
```

The Parser

Review the functional pearl paper by Hutton and Meijer (2).

```
newtype Parser a = Parser (String -> [(a, String)])

parse               :: Parser a -> String -> [(a, String)]
parse (Parser p) s   = p s

itemParser    :: Parser Char
itemParser     = Parser ( \ s -> case s of c:r -> [(c, r)]
                                           ""  -> [] )

consParser    :: a -> Parser a
consParser x   = Parser ( \ s -> [(x, s)] )

parse itemParser "this is to be parsed"
    ⟹    [('t', "his is to be parsed")]
```

The Parser

Review the functional pearl paper by Hutton and Meijer (2).

```
newtype Parser a = Parser (String -> [(a, String)])

parse               :: Parser a -> String -> [(a, String)]
parse (Parser p) s   = p s

itemParser    :: Parser Char
itemParser     = Parser ( \ s -> case s of c:r -> [(c, r)]
                                           ""  -> [] )

consParser    :: a -> Parser a
consParser x   = Parser ( \ s -> [(x, s)] )

parse (consParser (+)) "this will remain unchanged"
    ⟹    [((+), "this will remain unchanged")]
```

Parser Monad

```
joinParser     :: Parser a -> (a -> Parser b) -> Parser b
joinParser p q = Parser ( \ s -> concat [
                    parse (q r) t | (r, t) <- parse p s ] )
```

Parser Monad

```
joinParser     :: Parser a -> (a -> Parser b) -> Parser b
joinParser p q = Parser ( \ s -> concat [
                   parse (q r) t | (r, t) <- parse p s ] )

instance Monad Parser where     (>>=)    = joinParser
                                return   = consParser

pairParser :: Parser (Char, Char)
pairParser  = do x <- itemParser
                 y <- itemParser
                 return (x, y)
```

Parser Monad

```
joinParser     :: Parser a -> (a -> Parser b) -> Parser b
joinParser p q = Parser ( \ s -> concat [
                   parse (q r) t | (r, t) <- parse p s ] )

instance Monad Parser where     (>>=)    = joinParser
                                return   = consParser

pairParser :: Parser (Char, Char)
pairParser  = do x <- itemParser
                 y <- itemParser
                 return (x, y)

parse pairParser "this is to be parsed"
    ⟹    [(('t', 'h'), "is is to be parsed")]
```

Parser Monad

```
joinParser     :: Parser a -> (a -> Parser b) -> Parser b
joinParser p q = Parser ( \ s -> concat [
                   parse (q r) t | (r, t) <- parse p s ] )

instance Monad Parser where     (>>=)    = joinParser
                                return   = consParser

pairParser :: Parser (Char, Char)
pairParser  = do x <- itemParser
                 y <- itemParser
                 return (x, y)

parse (do pairParser; pairParser) "this is to be parsed"
    ⟹    [(('i', 's'), " is to be parsed")]
```

Parsing XML

```
data Element = Elem String [(String, String)] [Content]
data Content = CElem Element | CText String
```

Parsing XML

```
data Element = Elem String [(String, String)] [Content]
data Content = CElem Element | CText String

xmlelt :: Parser Element
xmlelt =  do (e, as) <- opening
             xs <- many (xmlelt +++ pcdata)
             closing e
             return (Elem e as xs)
```

Parsing XML

```
data Element = Elem String [(String, String)] [Content]
data Content = CElem Element | CText String

xmlelt :: Parser Element
xmlelt =  do (e, as) <- opening
             xs <- many (xmlelt +++ pcdata)
             closing e
             return (Elem e as xs)

closing, identify :: String -> Parser ()
closing e  = do char '<'; char '/'; identify e; char '>'
                return ()
identify e = case e of c:r -> do char c; identify r
                       ""  -> return ()
```

References

Ralf Hinze and Johan Jeuring.

Generic Haskell: Practice and Theory.

In Roland Backhouse and Jeremy Gibbons, editors, *Generic Programming: Advanced Lectures*, volume 2793 of *LNCS*, pages 1–56. Springer, 2003.

Graham Hutton and Erik Meijer.

Monadic Parsing in Haskell.

Journal of Functional Programming, 8(4), 1998.

Benjamin C. Pierce.

Types and Programming Languages.

MIT Press, Cambridge, MA, USA, 2002.