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

Type Classes

Classes

Polymorphism captures similar structures over different values, while type classes capture similar operations over different structures. (1)

Classes

Polymorphism captures similar structures over different values, while type classes capture similar operations over different structures. (1)

Type classes enrich the type system with function overloading and bring together ad-hoc vs. parametric polymorphism. They were introduced into Haskell by Wadler and Blott (3).

Classes

> *Polymorphism captures similar structures over different values, while type classes capture similar operations over different structures.* (1)

Type classes enrich the type system with function overloading and bring together ad-hoc vs. parametric polymorphism. They were introduced into Haskell by Wadler and Blott (3).

```
class Eq a where
    (==), (/=) :: a -> a -> Bool

    -- Minimal complete definition: (==) or (/=)
    x == y        = not (x /= y)
    x /= y        = not (x == y)
```

Qualified Types

Qualified types limit their polymorphism of to given type classes.

Qualified Types

Qualified types limit their polymorphism of to given type classes.

```
(==)  :: Eq a  => a -> a -> Bool
(+)   :: Num a => a -> a -> a
```

Qualified Types

Qualified types limit their polymorphism of to given type classes.

```
(==)  :: Eq a  => a -> a -> Bool
(+)   :: Num a => a -> a -> a

elem   :: Eq a => a -> [a] -> Bool
elem    = any . (==)

any    :: (a -> Bool) -> [a] -> Bool
any p   = or . map p
```

Qualified Types

Qualified types limit their polymorphism of to given type classes.

```haskell
(==) :: Eq a  => a -> a -> Bool
(+)  :: Num a => a -> a -> a

elem   :: Eq a => a -> [a] -> Bool
elem    = any . (==)

any    :: (a -> Bool) -> [a] -> Bool
any p   = or . map p

nubBy :: (a -> a -> Bool) -> [a] -> [a]
nub   :: Eq a => [a] -> [a]
```

nub ≡ nubBy (==)

Subclassing

```
class (Eq a, Show a) => Num a where
    (+), (-), (*)   :: a -> a -> a
    negate          :: a -> a
    abs, signum     :: a -> a

    -- Minimal complete definition: except negate or (-)
    x - y            = x + negate y
    negate x         = 0 - x
```

Subclassing

```
class (Eq a, Show a) => Num a where
    (+), (-), (*)    :: a -> a -> a
    negate           :: a -> a
    abs, signum      :: a -> a

    -- Minimal complete definition: except negate or (-)
    x - y             = x + negate y
    negate x          = 0 - x


class Eq a => Ord a where
    compare                   :: a -> a -> Ordering
    (<), (<=), (>=), (>)      :: a -> a -> Bool
    max, min                  :: a -> a -> a
```

```haskell
-- Minimal complete definition: (<=) or compare
compare x y | x == y    = EQ
            | x <= y    = LT
            | otherwise = GT

x <= y = compare x y /= GT
x < y  = compare x y == LT
x >= y = compare x y /= LT
x > y  = compare x y == GT

max x y   | x <= y      = y
          | otherwise   = x
min x y   | x <= y      = x
          | otherwise   = y
```

Instances

Type's membership into a class and the instances of methods can be either declared or derived.

```
data Tree a = Node a [Tree a]     deriving Show
```

Instances

Type's membership into a class and the instances of methods can be either declared or derived.

```
data Tree a = Node a [Tree a]


instance Show a => Show (Tree a) where show = showTree



showTree  (Node a t) = show a ++ "<"
                     ++ concat (map showTree t)
                     ++ ">"
```

Instances

Type's membership into a class and the instances of methods can be either declared or derived.

```
data Tree a = Node a [Tree a]

instance Show a => Show (Tree a) where showsPrec _ =
                                                showsTree

showsTree (Node a t) = shows a . ("<" ++)
                       . flip (foldr showsTree) t
                       . (">" ++)
```

Instances

Type's membership into a class and the instances of methods can be either declared or derived.

```
data Tree a = Node a [Tree a]

instance Show a => Show (Tree a) where showsPrec _ =
                                                   showsTree

showsTree (Node a t) = shows a . ("<" ++)
                     . flip (foldr showsTree) t
                     . (">" ++)

shows           :: Show a => a -> String -> String
shows            = showsPrec 0
```

Part II

Pretty-Printing

Pretty-Printing

Explore `Text.PrettyPrint` by Hughes and Peyton Jones, and compare it with the paper by Wadler (2) and Leijen's `PPrint`.

Pretty-Printing

Explore `Text.PrettyPrint` by Hughes and Peyton Jones, and compare it with the paper by Wadler (2) and Leijen's `PPrint`.

```
class Pretty a where    pretty :: a -> Doc
```

Pretty-Printing

Explore `Text.PrettyPrint` by Hughes and Peyton Jones, and compare it with the paper by Wadler (2) and Leijen's `PPrint`.

```
class Pretty a where    pretty :: a -> Doc


instance Show Doc where


    showsPrec _ = displayS . renderPretty 0.4 80
```

Pretty-Printing

Explore `Text.PrettyPrint` by Hughes and Peyton Jones, and compare it with the paper by Wadler (2) and Leijen's `PPrint`.

```
class Pretty a where    pretty :: a -> Doc

instance Show Doc where

    showsPrec _ = displayS . renderPretty 0.4 80

instance Pretty a => Pretty (Tree a) where

    pretty (Node a t) = pretty a <> text "<"
                      <> foldr ((<>) . pretty) empty t
                      <> text ">"
```

Pretty-Printing

Explore `Text.PrettyPrint` by Hughes and Peyton Jones, and compare it with the paper by Wadler (2) and Leijen's `PPrint`.

```
class Pretty a where    pretty :: a -> Doc

instance Show Doc where

    showsPrec _ = displayS . renderPretty 0.4 80

instance Pretty a => Pretty (Tree a) where

    pretty (Node a t) = pretty a <> encloseSep (text "<")
                                                (text ">")
                                          empty (map pretty t)
```

References

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

Philip Wadler.
A Prettier Printer.
In Jeremy Gibbons and Oege de Moor, editors, *The Fun of Programming*, Cornerstones of Computing, pages 223–243. Palgrave Macmillan, March 2003 2003.

Philip Wadler and Stephen Blott.
How to Make Ad-Hoc Polymorphism Less Ad Hoc.
In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 60–76. ACM, January 1989.