Lecture 10: Type Classes and Miscellaneous

Viliam Lisý

Artificial Intelligence Center
Department of Computer Science, Faculty of Electrical Eng.
Czech Technical University in Prague

viliam.lisy@fel.cvut.cz

April, 2020

Example: Arithmetic expressions

Recursive types can represent tree structures, such as **expressions** from numbers, plus, multiplication.

```
data Expr = Val Int
          | Add Expr Expr
          | Mul Expr Expr
```

1+2*3

```
Add (Val 1) (Mul (Val 2) (Val 3))
```

Example: Arithmetic expressions

Using recursion, it is now easy to define functions that process expressions. For example:

```
size :: Expr -> Int
size (Val n)   = 1
size (Add x y) = size x + size y
size (Mul x y) = size x + size y

eval :: Expr -> Int
eval (Val n)   = n
eval (Add x y) = eval x + eval y
eval (Mul x y) = eval x * eval y
```

Type Classes

Collection of types that can be used with the same functions `Eq`, `Ord`, `Show`. Functions required by a class can be accessed by

`:info <classname>`.

```
> :info Eq

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

Functions can often be implemented based on other. Only minimal complete definition (one of the above) is required.

Show Class

A class values convertible to a readable string

```
class Show a where
  showsPrec :: Int -> a -> ShowS
  show :: a -> String
  showList :: [a] -> ShowS
```

```
type ShowS = String -> String
```

This allows constant-time concatenation of results using function composition (optimization)

Minimal complete definition: `showsPrec` | `show`

Instance of a Class

A new instance can be added to a class by

```haskell
instance Show Nat where
  show n = "N" ++ show (nat2int n)
```

```haskell
instance Show Expr where
  show (Val n) = show n
  show (Add e1 e2) = "(+ " ++ show e1 ++ " "
                       ++ show e2  ++ ")"
  show (Mul e1 e2) = "(* " ++ show e1 ++ " "
                       ++ show e2  ++ ")"
```

Class Contexts

Remember the definition

```
data Maybe a = Nothing | Just a
```

To make Maybe an instance of Eq, a has to be in Eq

```
instance Eq a => Eq (Maybe a) where
    Nothing == Nothing = True
    (Just x) == (Just x') = x == x'
```

Obvious definition of instances are automated

```
data Shape = Circle Float
           | Rect Float Float
           deriving (Show, Eq)
```

The implemented function bodies determine the minimum required functions

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
    x == y = not (x /= y)
    x /= y = not (x == y)
```

Functor Class

Class of structures you can map over

```
class Mapable f where
      mmap :: (a -> b) -> f a -> f b
```

```
instance Mapable[] where
       mmap = map

instance Mapable Maybe where
       mmap f (Just x) = Just (f x)
       mmap f Nothing = Nothing
```

Kinds

Types of types and type constructors

`*`	A specific type
`* -> *`	A type that given a type creates a type
`Constraint`	A constructor of a type constraint
`:k`	

Types Summary

- Everything has a type known in compile time
  - basic values
  - functions
  - data structures
- Types are key for data structures in Haskell
- Types can be instances of classes
  - overloaded functions
- "Types" of types are kinds

Higher Order Functions

The same functions as in scheme are available

```
map :: (a -> b) -> [a] -> [b]
filter :: (a -> Bool) -> [a] -> [a]
```

```
map f xs = [f x | x <- xs]
filter p xs = [x | x <- xs, p x]
```

Foldr

```haskell
foldr :: (a -> b -> b) -> b -> [a] -> b
```

```
sum [1,2,3]
=
foldr (+) 0 [1,2,3]
=
foldr (+) 0 (1:(2:(3:[])))
=
1+(2+(3+0))
=
6
```

`fold fn el list` can be interpreted as:
replace each `(:)` by `‘fn‘` and `[]` by `el`.

Lambda Expressions

Functions can be constructed without naming the functions by using **lambda expressions**.

```
\x -> x + x
```

As in scheme,

```
add x y = x + y
```

means

```
add = \x -> (\y -> x + y)
```

We also have the automated currying

```
add = \x y -> x + y
```

Lambda Expressions

We can use lambda expressions and local functions interchangeably

```
odds n = map f [0..n-1]
         where
            f x = x*2 + 1
```

can be simplified to

```
odds n = map (\x -> x*2 + 1) [0..n-1]
```

The earlier may be better if the local function has a natural name

Operator Sections

An **infix** operator can be converted into a curried **prefix** function by using parentheses.

```
> (+) 1 2
3
```

This convention also allows one of the arguments of the operator to be included in the parentheses.

```
> (1+) 2
3
> (+2) 1
3
```

If $\oplus$ is an operator then $(\oplus)$, $(x\oplus)$ and $(\oplus y)$ are called sections.

Custom Data Constructors

Begin with :

`:#, :+, :::`

```haskell
infixr :+
data MList a = Empty | a :+ MList a deriving Show
```

Modules

Haskell program is a collection of modules

- name spaces, abstract data declarations
- module names start with upper-cased character
- filenames must match module names in GHC
- `module <name> ( <exported>, <symbols> ) where`
- without exported symbols, everything is exported
- data constructors exported with type name
- `Tree(Leaf,Branch)`, can be abbreviated to `Tree(..)`

Example Module

```haskell
module Tree ( Tree(Leaf,Branch), fringe ) where

data Tree a = Leaf a | Branch (Tree a) (Tree a)

fringe :: Tree a -> [a]
fringe (Leaf x)             = [x]
fringe (Branch left right) =
                   fringe left ++ fringe right
```

Importing Modules

Imports must be at the beginning of a module
Prelude module is loaded by default
We can choose names to import and hide

```
import Tree
import Tree hiding (tree1)
import Tree (tree1, fringe)
import qualified Tree as T hiding (tree1)
:m + Tree
```

Advanced Pattern Matching

Data constructors can be matched nested

`(1, (x:xs), 'a', (2, Just y:ys))`

but not `x:x:xs`

Top-down, left-right

Matching can succeed, fail, diverge

Refutable patterns: `[]`, `Tree x l r`

Irrefutable patterns: `_`, `x`, `a`, `~(x:xs)`.

Lazy Pattern

Lazy pattern $\sim$ pat is irrefutable (always matches)

- The variable pat is bound only when used
- `~(x:xs)` on LHS is equivalent to using head/tail on RHS
- `~(x,y)` on LHS is equivalent to using fst/snd on RHS

```
(\~(a,b) -> 1) bot
```

A new instance can be added to a class by

Case Expressions

```
f p11 ... p1k = e1
...
f pn1 ... pnk = en
```

where each `pij` is a pattern, is semantically equivalent to:

```
f x1 x2 ... xk = case (x1, ... , xk) of
    (p11, ..., p1k ) -> e1
     ...
    (pn1, ..., pnk ) -> en
```

Pattern Matching Divergence

Assume the infinite recursion `bot = bot`

Pattern matching diverges if it tries to match bot

Order of definitions influences pattern matching failure

```
take 0 _ = []
take _ [] = []
take n (x:xs) = x : take (n-1) xs
```

```
take1 _ [] = []
take1 0 _ = []
take1 n (x:xs) = x : take1 (n-1) xs
```

Summary

- Type and type classes essential for Haskell
- Unnecessary, but pleasant Haskell features
  - higher order functions
  - lambda functions
  - infix operator sections
  - modules