Programming in Haskell
Aug-Nov 2015

LECTURE 14

OCTOBER 1, 2015

S P Suresh
Chennai Mathematical Institute

Enumerated data types

- The `data` keyword is used to define new types
- `data Bool = False | True`
- `data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat`

Data types with parameters

- ```
  data Shape = Circle Float
             | Square Float
             | Rectangle Float Float
  ```

- `Circle 5.0, Square 4.0, Rectangle 3.0 4.0`
  ```

Functions on data types

- Functions can be defined using pattern matching

- ```
  weekend :: Day -> Bool
  weekend Sat = True
  weekend Sun = True
  weekend _   = False
  ```

- ```
  area :: Shape -> Float
  area (Circle r)        = pi*r*r
  area (Square x)        = x*x
  area (Rectangle l w)   = l*w
  where
       pi = 3.1415927
  ```

Functions on data types

- What about

  `weekend2 :: Day -> Bool`

- ```
  weekend2 d
      | (d == Sat || d == Sun) = True
      | otherwise              = False
  ```

- Error - No instance for `(Eq Day)` arising from a use of ‘`==`’
  ```

Functions on data types

- How about this function?

- ```
  nextday :: Day -> Day
  nextday Sun = Mon
  nextday Mon = Tue
  ...
  nextday Sat = Sun
  ```

- Invoking `nextday Fri` in ghci will lead to error

- Error - No instance for `(Show Day)` arising from a use of `'print'`
  ```

Add data types to typeclasses

- To check equality of two values of a data type `a`, `a` must belong the type class `Eq`
- We add `Day` to the type class `Eq` as follows
- ```
  data Day = Sun | Mon | ... | Sat
       deriving Eq
  ```
- Default behaviour: `Sun == Sun`, `Tue /= Fri`, ...
- Now `weekday2` compiles without error
  ```

The type class Show

- To make nextday work, we must make Day an instance of Show

- ```
  data Day = Sun | Mon | ... | Sat
       deriving (Eq, Show)
  ```

- The type class Show consists of all data types that implement the function show
  ```

More derivations

- `show` converts its input to a string which can be printed on the screen
- Default text representation
- `show Wed == "Wed"`
- ```
  data Day = Sun | Mon | ... | Sat
      deriving (Eq, Show, Ord)
  ```
- `Sun < Mon < ... < Sat`
  ```

More derivations ...

- ```
  data Shape = Circle Float
             | Square Float
             | Rectangle Float Float
      deriving (Eq, Ord, Show)
  ```

- `show (Circle 5.0) == "Circle 5.0"`

- ```
  Square 4.0 == Square 4.0
  Square 4.0 /= Square 3.0
  Circle 5.0 /= Rectangle 3.0 4.0
  ```

- `Square 4.0 > Circle 5.0`

Constructors

- `Square`, `Circle`, `Sun`, `Mon`, ... are constructors
- They are functions

```
Sun :: Day
Rectangle :: Float -> Float -> Shape
Circle :: Float -> Shape
```

Constructors ...

* Constructors can be used just like other functions
* `Circle 5.0 :: Shape`
* `map Circle :: [Float] -> [Shape]`
* `map Circle [3.0, 2.0] = [Circle 3.0, Circle 2.0]`

Records

- ```
  data Person = Person String Int Float String
          deriving Show
  ```
- ```
  guy = Person "Alpha" 21 5.8 "+914427470226"
  ```
- ```
  name :: Person -> String
  name (Person n _ _ _) = n
  ```
- ```
  age :: Person -> Int
  age (Person _ a _ _) = a
  ```

Records ...

- ```
  height :: Person -> Float
  height (Person _ _ h _) = h
  ```
- ```
  phone :: Person -> Int
  phone (Person _ _ _ p) = p
  ```

Record syntax

* ```
  data Person = Person { name :: String
                       , age :: Int
                       , height :: Float
                       , phone :: String
                       } deriving Show
  ```

* ```
  guy = Person {name="Alpha", age = 21,
  height = 5.8, phone = "+914427470226"}
  ```

* The field names are actually functions

* ```
  name :: Person -> String
  age :: Person -> Int
  ```
  ```

Summary

- The keyword `data` is used to declare new data types
- The keyword `deriving` to derive as an instance of a type class
- Data types with parameters - `Shape, Person`
- Sum type or union - `Day, Shape`
- Product type or struct - `Person`

Abstract data types

- Consider a `Stack` data type
  - a collection of `Int`s stacked one on top of the other
  - `push`: place an element on top of the stack
  - `pop`: remove the topmost element of the stack
- Behaviour similar to lists

Abstract data types

- `type Stack = [Int]`

- ```
  push :: Int -> Stack -> Stack
  push x s = x:s
  ```

- ```
  pop :: Stack -> (Int, Stack)
  pop (x:s') = (x, s')
  ```

- Internal representation is evident. `Stack` is just a synonym

- ```
  insert :: Int -> Int -> Stack -> Stack
  insert x n s = (take (n-1) s) ++ [x]
                             ++ (drop (n-1) s)
  ```
  ```

Abstract data types

* `data Stack = Stack [Int]`
* The value constructor `Stack` is a function that converts a list of `Int` to a `Stack` object
* Internal representation hidden

Abstract data types

- ```
  empty :: Stack
  empty = Stack []
  ```

- ```
  push :: Int -> Stack -> Stack
  push x (Stack xs) = Stack (x:xs)
  ```

- ```
  pop :: Stack -> (Int, Stack)
  pop (Stack (x:xs)) = (x, Stack xs)
  ```

- ```
  isempty :: Stack -> Bool
  isempty (Stack []) = True
  isempty (Stack _) = False
  ```

Type parameters

* Polymorphic user-defined data types
* ```
  data Stack a = Stack [a]
       deriving (Eq, Show, Ord)
  ```
* `empty :: Stack a`
* `push :: Int -> Stack a -> Stack a`
* `pop :: Stack a -> (a, Stack a)`
* `isempty :: Stack a -> Bool`
  ```

Type parameters...

- Suppose we want to sum all elements in a stack
- `sumStack (Stack xs) = sum xs`
- What is the type of `sumStack`?
- Applicable only if the stack has numeric elements
- `sumStack :: (Num a) => Stack a -> a`

A custom `show`

- `show (Stack [1,2,3]) == "Stack [1,2,3]"`
- deriving `Show` defines a default implementation for `show`
- Suppose we want something mildly fancy
- `show (Stack [1,2,3]) == "1->2->3"`

A custom show

- One can change the default behaviour

- ```
  printElems :: (Show a) => [a] -> String
  printElems [] = ""
  printElems [x] = show x
  printElems (x:xs) = show x ++ "->" ++
                             printElems xs
  ```

- ```
  instance (Show a) => Show (Stack a) where
      show (Stack l) = printElems l
  ```

Queues

- Consider a `Queue` data type
  - a collection of `Int`s arranged in a sequence
  - `enqueue`: add an element at the end of the queue
  - `dequeue`: remove the element at the start of the queue

Queues

- `data Queue a = Queue [a]`
- ```
  empty :: Queue a
  empty = Queue []
  ```
- ```
  isempty :: Queue a -> Bool
  isempty (Queue []) = True
  isempty (Queue _) = False
  ```

Queues

- ```
  enqueue :: a -> Queue a -> Queue a
  enqueue x (Queue xs) = Queue (xs ++ [x])
  ```
- ```
  dequeue :: Queue a -> (a, Queue a)
  dequeue (Queue (x:xs)) = (x, Queue xs)
  ```

Queues

- Each enque on a queue of length n takes $O(n)$ time
- Enqueueing and dequeueing n elements might take $O(n^2)$ time

Efficient queue

- Use two lists
- Represent $q_1, q_2, ..., q_n$ as
  $[q_1, q_2, ..., q_j]$ and $[q_n, q_{n-1}, ..., q_{j+1}]$
- Second list is the second part of queue in reversed order
- enqueue adds an element at the start of the second list
- dequeue removes an element from the start of the first list

Efficient queue

* What if we try to dequeue when the first list is empty?
* We reverse the second list into the first, and remove the first element

Efficient queue

- `data Queue a = NuQu [a] [a]`
- `enqueue x (NuQu ys zs) = NuQu ys (x:zs)`
- ```
  dequeue (NuQu (x:xs) ys) =
                    (x, NuQu xs ys)
  dequeue (NuQu [] ys) =
       dequeue (NuQu (reverse ys) [])
  ```
  ```

Efficient queue

- If we add n elements, we get a queue

  `NuQu [] [qn,qn-1,...,q1]`

  - Next dequeue takes O(n) time to reverse the list
  - After one dequeue we get `NuQu [q2,...,qn] []`
  - Next n-1 dequeue operations take O(1) time

Amortized analysis

- How many times is an element touched?
  - Once when it is added to the second list
  - Twice when it is moved from the second to first
  - Once when it is removed from the first list
- Each element is touched at most four times
- Any sequence of n instructions involves at most n elements
- So any sequence of n instructions takes only $O(n)$ steps

Summary

- Abstract data types
- We can define polymorphic user-defined data types by supplying type parameters
- Conditional polymorphism for functions defined on such data
- The `instance` keyword to define non-default implementation of functions
- Efficient queues and amortized analysis