Programming in Haskell: Lecture 20

S P Suresh

October 23, 2019

Abstract data types

- Recall the `Stack` data type

```
data Stack a = Stack [a]
push :: a -> Stack a -> Stack a
push x (Stack xs) = Stack (n:xs)
pop :: Stack a -> (a, Stack a)
pop (Stack (x:xs)) = (x, Stack xs)
empty :: Stack a
empty = Stack []
isEmpty :: Stack a -> Bool
isEmpty (Stack xs) = null xs
```

Abstract data types

- If `st1 :: Stack a` and `st2 :: Stack a` ...

Abstract data types

- If `st1 :: Stack a` and `st2 :: Stack a` …
- we cannot do `st1 ++ st2`

Abstract data types

- If `st1 :: Stack a` and `st2 :: Stack a` ...
- we cannot do `st1 ++ st2`
- But we can write the following function:

```
attach :: Stack a -> Stack a -> Stack a
attach (Stack xs) (Stack ys) = Stack (xs++ys)
```

Abstract data types

- If `st1 :: Stack a` and `st2 :: Stack a` ...
- we cannot do `st1 ++ st2`
- But we can write the following function:

```
attach :: Stack a -> Stack a -> Stack a
attach (Stack xs) (Stack ys) = Stack (xs++ys)
```

- So what have we gained by making it a data type?

Modules

- Ideally we would like the internal representation to be hidden from the world

Modules

- Ideally we would like the internal representation to be hidden from the world
- **Solution**: Create a `Stack` module

Modules

- Ideally we would like the internal representation to be hidden from the world
- **Solution**: Create a `Stack` module
- A **module** consists of functions that are related to each other

Modules

- Ideally we would like the internal representation to be hidden from the world
- **Solution**: Create a `Stack` module
- A **module** consists of functions that are related to each other
- The name of the file must match the name of the module

Modules

- Ideally we would like the internal representation to be hidden from the world
- **Solution**: Create a `Stack` module
- A **module** consists of functions that are related to each other
- The name of the file must match the name of the module
- The module can be used (imported) in any other file in the same directory

A Stack *module*

- The Stack module, saved in Stack.hs

```haskell
module Stack(push, pop, empty, isEmpty) where

data Stack a = Stack [a]

push x (Stack xs) = Stack (x:xs)
pop (Stack (x:xs)) = (x, Stack xs)
empty = Stack []
isEmpty (Stack xs) = null xs
```

A `Stack` module

- The `Stack` module, saved in `Stack.hs`

```
module Stack(push, pop, empty, isEmpty) where

data Stack a = Stack [a]

push x (Stack xs) = Stack (x:xs)
pop (Stack (x:xs)) = (x, Stack xs)
empty = Stack []
isEmpty (Stack xs) = null xs
```

- The functions listed inside parentheses can be used outside the module

Using the Stack *module*

- The following code is in `postfix.hs`, in the same directory

```
import Stack


myStack = empty
myStack' = push 5 myStack
```

Using the Stack *module*

- The following code is in `postfix.hs`, in the same directory

```haskell
import Stack


myStack = empty
myStack' = push 5 myStack
```

- Does not compile!

```
-- Not in scope: type constructor or class 'Stack'
```

Using the Stack module

- Need to export the type constructor

```
module Stack(Stack, push, pop, empty, isEmpty) where
data Stack a = Stack [a]
...
```

Using the Stack *module*

- Need to export the type constructor

```haskell
module Stack(Stack, push, pop, empty, isEmpty) where
data Stack a = Stack [a]
...
```

- Now `postfix.hs` compiles

```haskell
import Stack


myStack = empty
myStack' = push 5 myStack
```

Using the `Stack` *module*

- Can we do this in `postfix.hs`?

```
import Stack
newStack = Stack [0..9]
```

Using the `Stack` *module*

- Can we do this in `postfix.hs`?

```haskell
import Stack
newStack = Stack [0..9]
```

- Does not compile!

```haskell
-- Data constructor not in scope: Stack :: [Integer] -> t
```

Using the `Stack` *module*

- Can we do this in `postfix.hs`?

```
import Stack
newStack = Stack [0..9]
```

- Does not compile!

```
-- Data constructor not in scope: Stack :: [Integer] -> t
```

- This is exactly what we want!

Using the `Stack` *module*

- Can we do this in `postfix.hs`?

```
import Stack
newStack = Stack [0..9]
```

- Does not compile!

```
-- Data constructor not in scope: Stack :: [Integer] -> t
```

- This is exactly what we want!
- No one should be able to directly use the data constructor!

Exporting the constructors

- If we want the data constructors to be used directly …

Exporting the constructors

- If we want the data constructors to be used directly …
- we export the data constructors in the module

```
module Stack(Stack(Stack), push, pop, empty, isEmpty) where
data Stack a = Stack [a]
    ...
```

Exporting the constructors

- If we want the data constructors to be used directly …
- we export the data constructors in the module

```
module Stack(Stack(Stack), push, pop, empty, isEmpty) where
data Stack a = Stack [a]
    ...
```

- In `Stack(Stack)`, the left `Stack` is the export of the type constructor

Exporting the constructors

- If we want the data constructors to be used directly …
- we export the data constructors in the module

```
module Stack(Stack(Stack), push, pop, empty, isEmpty) where
data Stack a = Stack [a]
   ...
```

- In `Stack(Stack)`, the left `Stack` is the export of the type constructor
- The right `Stack` is the data constructor

Exporting the constructors

- If we want the data constructors to be used directly …
- we export the data constructors in the module

```
module Stack(Stack(Stack), push, pop, empty, isEmpty) where
data Stack a = Stack [a]
   ...
```

- In `Stack(Stack)`, the left `Stack` is the export of the type constructor
- The right `Stack` is the data constructor
- In case there are many data constructors, we export them all by:

```
module Stack(Stack(..), push, pop, empty, isEmpty) where
```

Using the `Stack` module

- We create `module Stack` in `Stack.hs`

Using the `Stack` module

- We create `module Stack` in `Stack.hs`
  - exporting the type constructor but not the data constructor

Using the `Stack` module

- We create `module Stack` in `Stack.hs`
  - exporting the type constructor but not the data constructor
- We create `postfix.hs`, importing `Stack`

Using the `Stack` module

- We create `module Stack` in `Stack.hs`
  - exporting the type constructor but not the data constructor
- We create `postfix.hs`, importing `Stack`
- If we load `postfix.hs` in `ghci`, `Stack.hs` is also compiled

Using the `Stack` module

- We create `module Stack` in `Stack.hs`
  - exporting the type constructor but not the data constructor
- We create `postfix.hs`, importing `Stack`
- If we load `postfix.hs` in `ghci`, `Stack.hs` is also compiled
- But we can only use the exported functions

Using the `Stack` module

- We create `module Stack` in `Stack.hs`
  - exporting the type constructor but not the data constructor
- We create `postfix.hs`, importing `Stack`
- If we load `postfix.hs` in `ghci`, `Stack.hs` is also compiled
- But we can only use the exported functions
- One flaw in hiding our internal representation:

```
show (push 5 empty) = "Stack [5]"
```

Using the `Stack` *module*

- We create `module Stack` in `Stack.hs`
  - exporting the type constructor but not the data constructor
- We create `postfix.hs`, importing `Stack`
- If we load `postfix.hs` in `ghci`, `Stack.hs` is also compiled
- But we can only use the exported functions
- One flaw in hiding our internal representation:

```
show (push 5 empty) = "Stack [5]"
```

- There is a need for a custom **`show`**

A custom `show`

- Our original definition of `Stack`:

```
data Stack a = Stack [a]
    deriving (Eq, Ord, Show)
```

A custom show

- Our original definition of `Stack`:

```
data Stack a = Stack [a]
    deriving (Eq, Ord, Show)
```

- But the default **`show`** reveals the internal structure

A custom show

- Our original definition of `Stack`:

```
data Stack a = Stack [a]
    deriving (Eq, Ord, Show)
```

- But the default **show** reveals the internal structure
- We create a custom `Show` instance of `Stack a` as follows:

```
data Stack a = Stack [a]
    deriving (Eq, Ord)
instance Show a => Show (Stack a) where
    show (Stack l) = fancyShow l
```

A custom `show`

- Suppose we want `show (Stack [1,2,3]) = "1->2->3"`

A custom show

- Suppose we want `show (Stack [1,2,3]) = "1->2->3"`
- We create a custom `Show` instance as follows:

```
data Stack a = Stack [a]
    deriving (Eq, Ord)
instance Show a => Show (Stack a) where
    show (Stack l) = fancyShow l
fancyShow :: Show a => [a] -> String
fancyShow = (intercalate "->") . (map show)
```

Using `Stack` *– postfix expressions*

- A **postfix expression** is an arithmetic expression where the operator appears after the operands

Using Stack *– postfix expressions*

- A **postfix expression** is an arithmetic expression where the operator appears after the operands
- No parentheses required in a postfix expression

```
3 5 8 * + = (3 + (5 * 8)) = 43
2 3 + 7 2 + - = ((2 + 3) - (7 + 2)) = (-4)
```

Using Stack *– postfix expressions*

- Every postfix expression can be converted uniquely to an infix expression

Using Stack *– postfix expressions*

- Every postfix expression can be converted uniquely to an infix expression
- Start with an empty stack of expressions

Using Stack *– postfix expressions*

- Every postfix expression can be converted uniquely to an infix expression
- Start with an empty stack of expressions
- Scan the postfix expression from the left

Using Stack *– postfix expressions*

- Every postfix expression can be converted uniquely to an infix expression
- Start with an empty stack of expressions
- Scan the postfix expression from the left
- If the symbol is a number, it is a standalone expression

Using Stack *– postfix expressions*

- Every postfix expression can be converted uniquely to an infix expression
- Start with an empty stack of expressions
- Scan the postfix expression from the left
- If the symbol is a number, it is a standalone expression
    - Push it on to the stack

Using Stack – *postfix expressions*

- Every postfix expression can be converted uniquely to an infix expression
- Start with an empty stack of expressions
- Scan the postfix expression from the left
- If the symbol is a number, it is a standalone expression
    - Push it on to the stack
- If the symbol is an operator, bracket it with the top two expressions on stack

Using Stack – *postfix expressions*

- Every postfix expression can be converted uniquely to an infix expression
- Start with an empty stack of expressions
- Scan the postfix expression from the left
- If the symbol is a number, it is a standalone expression
  - Push it on to the stack
- If the symbol is an operator, bracket it with the top two expressions on stack
  - Pop the top two and push the result on to stack

Using `Stack` *– evaluating postfix expressions*

- Use the same logic described earlier

Using Stack *– evaluating postfix expressions*

- Use the same logic described earlier
- Start with an empty stack of numbers

Using Stack *– evaluating postfix expressions*

- Use the same logic described earlier
- Start with an empty stack of numbers
- Scan the postfix expression from the left

Using Stack *– evaluating postfix expressions*

- Use the same logic described earlier
- Start with an empty stack of numbers
- Scan the postfix expression from the left
- If the symbol is a number, push it on to the stack

Using Stack *– evaluating postfix expressions*

- Use the same logic described earlier
- Start with an empty stack of numbers
- Scan the postfix expression from the left
- If the symbol is a number, push it on to the stack
- If the symbol is an operator

Using Stack *– evaluating postfix expressions*

- Use the same logic described earlier
- Start with an empty stack of numbers
- Scan the postfix expression from the left
- If the symbol is a number, push it on to the stack
- If the symbol is an operator
  - pop the top two numbers on the stack

Using Stack *– evaluating postfix expressions*

- Use the same logic described earlier
- Start with an empty stack of numbers
- Scan the postfix expression from the left
- If the symbol is a number, push it on to the stack
- If the symbol is an operator
    - pop the top two numbers on the stack
    - apply operation

Using Stack – evaluating postfix expressions

- Use the same logic described earlier
- Start with an empty stack of numbers
- Scan the postfix expression from the left
- If the symbol is a number, push it on to the stack
- If the symbol is an operator
  - pop the top two numbers on the stack
  - apply operation
  - push the result on to stack

A calculator program

- A postfix expression is a sequence of numbers and operators

A calculator program

- A postfix expression is a sequence of numbers and operators
- We represent it as a list of tokens

```
import Stack
import Data.List (foldl')

data Token = Val Int | Op Char
type Expr = [Token]
```

Evaluating expressions

```
step :: Stack Int -> Token -> Stack Int
step st (Val n) = push n st
step st (Op c)
    | c == '+'  = push (n2+n1) st2
    | c == '-'  = push (n2-n1) st2
    | c == '*'  = push (n2*n1) st2
    | c == '/'  = push (n2 `div` n1) st2
    where (n1, st1) = pop st
          (n2, st2) = pop st1
eval :: Expr -> Int
eval = fst . pop . (foldl' step empty)
```

A calculator program

- Not convenient to provide input of the form `[Val 2, Val 3, Op '+']`

A calculator program

- Not convenient to provide input of the form `[Val 2, Val 3, Op '+']`
- Need a translator from strings to expressions (assuming only "correct" strings as input)

```
toExpr :: String -> Expr
toExpr str = map tokenize (words str)
tokenize :: String -> Token
tokenize "+" = Op '+'
tokenize "-" = Op '-'
tokenize "*" = Op '*'
tokenize "/" = Op '/'
tokenize str = Val (read str::Int)
```

A calculator program

- We can even make the program interactive

```
eval :: String -> Int
eval str = fst $ pop $ foldl' step empty (toExpr str)
main :: IO ()
main = interact (unlines . map (show . eval) . lines)
```

A calculator program

- We can even make the program interactive

```
eval :: String -> Int
eval str = fst $ pop $ foldl' step empty (toExpr str)
main :: IO ()
main = interact (unlines . map (show . eval) . lines)
```

- Add lines like these to `postfix.in`

```
22 34 +
2 5 + 8 *
2 5 8 + *
```

A calculator program

- We can even make the program interactive

```
eval :: String -> Int
eval str = fst $ pop $ foldl' step empty (toExpr str)
main :: IO ()
main = interact (unlines . map (show . eval) . lines)
```

- Add lines like these to `postfix.in`

```
22 34 +
2 5 + 8 *
2 5 8 + *
```

- Compile and run `./postfix < postfix.in` to see the results

A `Queue` *module*

- In a stack, elements are pushed and popped at the top

A Queue *module*

- In a stack, elements are pushed and popped at the top
- In a queue, elements are added at the rear and removed from the head

A Queue *module*

- In a stack, elements are pushed and popped at the top
- In a queue, elements are added at the rear and removed from the head
- The `Queue` module, saved in `Queue.hs`

```
module Queue(Queue, enqueue, dequeue, empty, isEmpty) where

data Queue a = Queue [a]
    deriving (Eq, Ord)
enqueue x (Queue xs) = Queue (xs++[x])
dequeue (Queue (x:xs)) = (x, Queue xs)
empty = Queue []
isEmpty (Queue xs) = null xs
```

A Queue *module*

- Each enqueue on a queue of length n takes $O(n)$ time

A Queue module

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

A `Queue` module

- Each `enqueue` on a queue of length n takes $O(n)$ time
- Enqueueing and dequeueing n elements might take $O(n^2)$ time
- A more efficient queue can be built by using two lists `front` and `back`

A `Queue` module

- Each `enqueue` on a queue of length n takes $O(n)$ time
- Enqueueing and dequeueing n elements might take $O(n^2)$ time
- A more efficient queue can be built by using two lists `front` and `back`
- `queue == front ++ reverse back`

A `Queue` *module*

- Each `enqueue` on a queue of length n takes $O(n)$ time
- Enqueueing and dequeueing n elements might take $O(n^2)$ time
- A more efficient queue can be built by using two lists `front` and `back`
- `queue == front ++ reverse back`
- To enqueue, add an element to the head of `back`

A `Queue` *module*

- Each `enqueue` on a queue of length n takes $O(n)$ time
- Enqueueing and dequeueing n elements might take $O(n^2)$ time
- A more efficient queue can be built by using two lists `front` and `back`
- `queue == front ++ reverse back`
- To enqueue, add an element to the head of `back`
- To dequeue, remove an element from the head of `front`

A `Queue` *module*

- Each `enqueue` on a queue of length n takes $O(n)$ time
- Enqueueing and dequeueing n elements might take $O(n^2)$ time
- A more efficient queue can be built by using two lists `front` and `back`
- `queue == front ++ reverse back`
- To enqueue, add an element to the head of `back`
- To dequeue, remove an element from the head of `front`
    - What if `front` is empty?

A `Queue` *module*

- Each `enqueue` on a queue of length n takes $O(n)$ time
- Enqueueing and dequeueing n elements might take $O(n^2)$ time
- A more efficient queue can be built by using two lists `front` and `back`
- `queue == front ++ reverse back`
- To enqueue, add an element to the head of `back`
- To dequeue, remove an element from the head of `front`
    - What if `front` is empty?
    - Reverse `back` into `front` and dequeue

A Queue module

- Efficient queue

```
module Queue(Queue, enqueue, dequeue, empty, isEmpty,
                              fromList, toList) where

data Queue a = Queue [a] [a]
    deriving (Eq, Ord)
instance Show a => Show (Queue a) where
    show q = "{" ++ show (toList q) ++ "}"

fromList l = Queue (l, [])
toList (Queue f b) = f ++ reverse b
```

A Queue module

- Efficient queue

```
module Queue(Queue, enqueue, dequeue, empty, isEmpty,
                              fromList, toList) where
....
enqueue x (Queue f b) = Queue f (x:b)

dequeue (Queue []    b) = dequeue (Queue (reverse b) [])
dequeue (Queue (x:f) b) = (x, Queue f b)

empty = Queue [] []
isEmpty (Queue f b) = null f && null b
```

Amortized analysis

- If we add `n` elements, we get a queue

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

Amortized analysis

- If we add `n` elements, we get a queue

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

- The next dequeue takes $O(n)$ time to reverse the list

Amortized analysis

- If we add `n` elements, we get a queue

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

- The next dequeue takes $O(n)$ time to reverse the list
- After one dequeue we get:

```
Queue [q2,...,qn] []
```

Amortized analysis

- If we add `n` elements, we get a queue

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

- The next dequeue takes $O(n)$ time to reverse the list
- After one dequeue we get:

```
Queue [q2,...,qn] []
```

- Next `n-1` dequeue operations take $O(1)$ time

Amortized analysis

- How many times is an element touched?

Amortized analysis

- How many times is an element touched?
  - Once when it is added to the second list

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

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

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

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

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

Applying queues – an ancient Telugu riddle

15 brahmins and 15 thieves had to spend a dark night at an isolated temple of *Durga*. At midnight, the Goddess appeared in person and wanted to devour just 15 persons because She was hungry. The thieves naturally suggested that She should eat the 15 soft-limbed brahmins. But the brahmins proposed that all the 30 would stand in a circle and that Durga should eat each ninth person. The proposal was accepted by Durga and the thieves. So the brahmins arranged themselves and the thieves in a circle, telling each one where to stand. Durga counted out each ninth person and devoured him. When 15 were thus eaten, She was satiated and disappeared, and only brahmins now remained in the circle.

How do you arrange the brahmins and thieves in the circle?

The Vanadurga riddle

- Imagine a circle with a "current position"

The Vanadurga riddle

- Imagine a circle with a "current position"
- The people starting from that position can be listed in clockwise order

The Vanadurga riddle

- Imagine a circle with a "current position"
- The people starting from that position can be listed in clockwise order
- The person in the current position would be at the head of the list

The Vanadurga riddle

- Imagine a circle with a "current position"
- The people starting from that position can be listed in clockwise order
- The person in the current position would be at the head of the list
- The person to the right would be next in the list, and so on

The Vanadurga riddle

- Imagine a circle with a "current position"
- The people starting from that position can be listed in clockwise order
- The person in the current position would be at the head of the list
- The person to the right would be next in the list, and so on
- The person to the left would be the last element of the list

The Vanadurga riddle

- What if we move one step clockwise? What is the list representing the new configuration?

The Vanadurga riddle

- What if we move one step clockwise? What is the list representing the new configuration?
- Instead of the position moving right, you can think of the list moving left

The Vanadurga riddle

- What if we move one step clockwise? What is the list representing the new configuration?
- Instead of the position moving right, you can think of the list moving left
- The previous head is now the tail

```
moveRight (x:xs) = xs ++ [x]
```

The Vanadurga riddle

- What if we move one step clockwise? What is the list representing the new configuration?
- Instead of the position moving right, you can think of the list moving left
- The previous head is now the tail

```
moveRight (x:xs) = xs ++ [x]
```

- Can use efficient queues to avoid the costly `(++)` operator

```
moveRight q = let (x,q') = dequeue q in enqueue q' x
```

The Vanadurga riddle – full solution

```
import Queue
-- Assume m >= 2, r < n, r >= 0
-- In the Vanadurga example, m = 9, r = 15, n = 30
vanadurga m r n = kill m r n (fromList [1..n], empty)
kill m r n (surv, dead)
    | r == 0    = (surv, dead)
    | otherwise = kill m (r-1) (n-1) $
                  shift (m-1 `mod` n) (surv, dead)
shift n (surv,dead)
    | n == 0        = (surv', enqueue x dead)
    | otherwise     = shift (n-1) (enqueue x surv', dead)
    where (x,surv') = dequeue surv
```

Summary

- Modules to hide implementation details

Summary

- Modules to hide implementation details
- Exporting type constructors, but hiding data constructors

Summary

- Modules to hide implementation details
- Exporting type constructors, but hiding data constructors
- The `instance` keyword to create custom instances

Summary

- Modules to hide implementation details
- Exporting type constructors, but hiding data constructors
- The `instance` keyword to create custom instances
- Using a stack to build a postfix calculator

Summary

- Modules to hide implementation details
- Exporting type constructors, but hiding data constructors
- The `instance` keyword to create custom instances
- Using a stack to build a postfix calculator
- Efficient queues and amortized analysis