AC21007: Haskell Lecture 7
Quick Sort, Monadic IO

František Farka

Recapitulation

- Tail recursion
  - Sum
  - Fibonacci numbers
  - Tail recursion and folds
- Algebraic data types
- (Light introduction to) Typeclasses

Quick Sort: Intuition

1. Choose an element in a list as “pivot”
2. Move all the elements larger than pivot to its right.
3. Move all the elements smaller than pivot to its left.
4. Recursively sort elements on left and on right of the pivot

Quick Sort in Haskell

- Quick Sort has two nice aspects:
  - Divide and Conquer
  - In-place sort

Quick Sort in Haskell

- Quick Sort has two nice aspects:
  - Divide and Conquer
  - In-place sort
- In-place sort like quick sort requires mutable arrays and mutable variables.

Quick Sort in Haskell

- Quick Sort has two nice aspects:
  - Divide and Conquer
  - In-place sort
- In-place sort like quick sort requires mutable arrays and mutable variables.
- To get pure version of quick sort, we need to forget about swapping, indexing, mutation.

Quick Sort in Haskell

- Quick Sort has two nice aspects:
  - Divide and Conquer
  - In-place sort
- In-place sort like quick sort requires mutable arrays and mutable variables.
- To get pure version of quick sort, we need to forget about swapping, indexing, mutation.
- Think in terms of creating new list based on input list.

Quick Sort in Haskell (cont.)

- How to pick a pivot?

Quick Sort in Haskell (cont.)

- How to pick a pivot? Take the first element.

Quick Sort in Haskell (cont.)

- How to pick a pivot? Take the first element.
- Sort a list:

```
quickSort [] = []
quickSort (x:xs) =
        let (left, right) = partition xs x
        in quickSort left ++ [x] ++ quickSort right
    where
        partition []     _  = ([], [])
        partition (y:ys) z  =
            let (vs, ws) = partition ys
            in if (y < z)
                    then (y:vs, ws)
                    else (vs, y:ws)
```

Quick Sort in Haskell (cont.)

- Quick Sort has two nice aspects:
  - Divide and Conquer
  - In-place sort
- Our version only demonstrate the divide and conquer part.
- Worst case time complexity: $\mathcal{O}(n^2)$
- Average time complexity: $\mathcal{O}(n \log n)$

Syntactic Intermezzo: case expression

- We saw ADTs
- How do we inspect values of ADTs?
  - Pattern matching in function definition
  - case expression

Syntactic Intermezzo: case expression

- We saw ADTs
- How do we inspect values of ADTs?
  - Pattern matching in function definition
  - case expression
- Syntax of case expression:

```
case <expr> of
    <pat_1>      ->  <expr_1>
    ...
    <pat_n>      ->  <expr_n>
```

$< expr_1 >$ to $< expr_n >$ are of some type a, the case expression has a value of the type a, e.g.:

```
case (safeHead someList) of
    Nothing -> "No head"
    Just h  -> "The head is: " ++ show h
```

Maybe as a monadic computation

- We saw the `Maybe` data type
- We saw that we can use it to enrich a range of a function (e. g. to make a partial function total):

Maybe as a monadic computation

- We saw the `Maybe` data type
- We saw that we can use it to enrich a range of a function (e. g. to make a partial function total):

```
head :: [a] -> a
head []      = error "Empty list"
head (x:_)   = x
```

vs.

```
safeHead :: [a] -> Maybe a
safeHead []      = Nothing
safeHead (x:_)   = Just x
```

Maybe as a monadic computation

- We saw the `Maybe` data type
- We saw that we can use it to enrich a range of a function (e. g. to make a partial function total):

```
head :: [a] -> a
head []      = error "Empty list"
head (x:_)   = x
```

vs.

```
safeHead :: [a] -> Maybe a
safeHead []      = Nothing
safeHead (x:_)   = Just x
```

- We will call `Maybe` is such a situation a *context* of a computation

Maybe as a monadic computation (cont.)

- Lets see how composable this approach is:

```
sqrtHead :: [Float] -> Float
sqrtHead xs = sqrt (head xs)
```

  - `head` fails on an empty list
  - `sqrt` fails on a negative number

Maybe as a monadic computation (cont.)

- Lets see how composable this approach is:

```
sqrtHead :: [Float] -> Float
sqrtHead xs = sqrt (head xs)
```

  - head fails on an empty list
  - sqrt fails on a negative number

- We already have safeHead, can we provide safeSqrt?

```
safeSqrt :: Float -> Maybe Float
safeSqrt a = if a < 0
                then Nothing
                else Just (sqrt a)
```

Maybe as a monadic computation (cont.)

- Lets see how composable this approach is:

```
sqrtHead :: [Float] -> Float
sqrtHead xs = sqrt (head xs)
```

  - head fails on an empty list
  - sqrt fails on a negative number
- We already have safeHead, can we provide safeSqrt?

```
safeSqrt :: Float -> Maybe Float
safeSqrt a = if a < 0
                then Nothing
                else Just (sqrt a)
```

- Let's compose these two into safeSqrtHead . . .

Maybe as a monadic computation (cont.)

- Lets see how composable this approach is:

```
sqrtHead :: [Float] -> Float
sqrtHead xs = sqrt (head xs)

safeSqrtHead :: [Float] -> Maybe Float
safeSqrtHead xs = case safeHead xs of
    Nothing     -> Nothing
    Just x      -> safeSqrt x
```

- Note the type signatures:

```
safeHead :: [a] -> Maybe a
safeSqrt :: Float -> Maybe Float
```

Maybe as a monadic computation (cont.)

- Lets see how composable this approach is:

```
sqrtHead :: [Float] -> Float
sqrtHead xs = sqrt (head xs)

safeSqrtHead :: [Float] -> Maybe Float
safeSqrtHead xs = case safeHead xs of
    Nothing     -> Nothing
    Just x      -> safeSqrt x
```

...the explicit case is verbose

- Note the type signatures:

```
safeHead :: [a] -> Maybe a
safeSqrt :: Float -> Maybe Float
```

Maybe as a monadic computation (cont.)

- Lets see how composable this approach is:

```
safeSqrtHead ::  [Float] -> Maybe Float
safeSqrtHead xs = safeHead xs `bind` safeSqrt

bind ::  Maybe Float -> (Float -> Maybe Float)
    -> Maybe Float
bind mval func = case mval of
        Nothing      -> Nothing
        Just val     -> func val
```

- Note the type signatures:

```
safeHead :: [a] -> Maybe a
safeSqrt :: Float -> Maybe Float
```

Maybe as a monadic computation (cont.)

- Lets see how composable this approach is:

```
safeSqrtHead ::  [Float] -> Maybe Float
safeSqrtHead xs = safeHead xs ‘bind‘ safeSqrt

bind ::  Maybe Float -> (Float -> Maybe Float)
    -> Maybe Float
bind mval func = case mval of
        Nothing     -> Nothing
        Just val    -> func val
```

…What is the most generic type of `bind`?

- Note the type signatures:

```
safeHead :: [a] -> Maybe a
safeSqrt :: Float -> Maybe Float
```

Maybe as a monadic computation (cont.)

- Lets see how composable this approach is:

```
safeSqrtHead ::  [Float] -> Maybe Float
safeSqrtHead xs = safeHead xs `bind` safeSqrt
```

```
bind ::  Maybe a -> (a -> Maybe b) -> Maybe b
bind mval func = case mval of
        Nothing      -> Nothing
        Just val     -> func val
```

...What is the most generic type of `bind`?

- Note the type signatures:

```
safeHead :: [a] -> Maybe a
safeSqrt :: Float -> Maybe Float
```

Monad typeclass

- We can abstract this technique over different data types using a typeclass (think of data types being “bindable” in the same way as being “orderable” and `Ord` typeclass)

Monad typeclass

- We can abstract this technique over different data types using a typeclass (think of data types being "bindable" in the same way as being "orderable" and `Ord` typeclass)
- The `Monad` t.c. as an interface for binding computations:

```
class Monad m where
    -- an operator instead of our ‘bind‘
    (>>=) :: m a -> (a -> m b) -> m b
    return :: a -> m a

instance Monad Maybe where
    Nothing    >>= _   = Nothing
    (Just a)   >>= f   = f a
    return a           = Just a
```

The `return` fnct to embed a pure value into a context

Monad typeclass

- We can abstract this technique over different data types using a typeclass (think of data types being "bindable" in the same way as being "orderable" and `Ord` typeclass)
- The `Monad` t.c. as an interface for binding computations:

```
class Monad m where
    -- an operator instead of our `bind`
    (>>=) :: m a -> (a -> m b) -> m b
    return :: a -> m a

instance Monad Maybe where
    Nothing     >>= _    = Nothing
    (Just a)    >>= f    = f a
    return a             = Just a
```

  The `return` fnct to embed a pure value into a context
- And our previous use case:

```
safeSqrtHead xs = safeHead xs >>= safeSqrt
sqrtOfTwo = return 2 >>= safeSqrt
```

Unit Data type - ()

- In Haskell all functions return a value
- Sometimes, we are not interested in the actual value
- There is a data type for this — `()` (unit) — that has a single constructor—also `()`.

Monadic IO

- In Haskell all IO happens in a context of type `IO a`
- `IO` encapsulates a state of the real world, you cannot construct or inspect values of this type directly

Monadic IO

- In Haskell all IO happens in a context of type `IO a`
- `IO` encapsulates a state of the real world, you cannot construct or inspect values of this type directly
- There are functions that take or return `IO` values:
  - `putStr, putStrLn :: String -> IO ()`
  - `getLine :: IO String`

Monadic IO

- In Haskell all IO happens in a context of type `IO a`
- `IO` encapsulates a state of the real world, you cannot construct or inspect values of this type directly
- There are functions that take or return `IO` values:
  - `putStr, putStrLn :: String -> IO ()`
  - `getLine :: IO String`
- And there is a `Monad IO` instance—`IO` computation can be sequenced using bind (>>=), a pure value can be injected into an `IO` context using `return`:

```
helloYou = getLine >>= \x ->
           putStrLn ("Hello " ++ x)
```

Monadic IO

- In Haskell all IO happens in a context of type `IO a`
- `IO` encapsulates a state of the real world, you cannot construct or inspect values of this type directly
- There are functions that take or return `IO` values:
  - `putStr, putStrLn :: String -> IO ()`
  - `getLine :: IO String`
- And there is a `Monad IO` instance—`IO` computation can be sequenced using bind ($>>=$), a pure value can be injected into an `IO` context using `return`:

```
helloYou =  getLine >>= \x ->
            putStrLn ("Hello " ++ x)
```

- We also say that there is an *effect*, which is performed in a monadic context (in general, not only `IO`).

Do Notation

- There is a syntax for monadic computations — `do` notation

Do Notation

- There is a syntax for monadic computations — `do` notation
- We call a single call to a function that returns monadic value an *action*. We either bind a value **in** this context to a variable:

  $\texttt{var}_n$ `<-` $\texttt{action}_n$

  or we ignore this value (we are interested only in the effect)

  $\texttt{action}_n$

Do Notation

- There is a syntax for monadic computations — `do` notation
- We call a single call to a function that returns monadic value an *action*. We either bind a value **in** this context to a variable:

  $\texttt{var}_n$ `<-` $\texttt{action}_n$

  or we ignore this value (we are interested only in the effect)

  $\texttt{action}_n$

  and we sequence such actions in a block, while using bound variables as arguments of other actions (following the action that binds the variable):

  ```
  do
      var1 <- action1
      var2 <- action2
      ...
      actionn vari varj
  ```

Do Notation

- There is a syntax for monadic computations — `do` notation
- We call a single call to a function that returns monadic value an *action*. We either bind a value **in** this context to a variable:

  $\texttt{var}_n$ `<-` $\texttt{action}_n$

  or we ignore this value (we are interested only in the effect)

  $\texttt{action}_n$

  and we sequence such actions in a block, while using bound variables as arguments of other actions (following the action that binds the variable):

```
do
    var1 <- action1
    var2 <- action2
    ...
    actionn vari varj
```

  the result of a `do` block is the result of last action (this action must not be a binding of a variable)

IO – A Simple Example

- A simple example of IO:

```
-- | Prompts a user for a number
getNumber :: String -> IO Int
getNumber username = do
          putStrLn ("Hello " ++ username ++ "!"
            ++ "Choose your favourite number:")
          x <- getLine
          putStrLn "Thank you!"
          return (read x)
```

An overview of IO functions

putChar :: `Char -> IO ()`
Write a character to the standard output device

putStr :: `String -> IO ()`
Write a string to the standard output device

putStrLn :: `String -> IO ()`
The same as putStr, but adds a newline character.

getChar :: `IO Char`
Read a character from the standard input device

getLine :: `IO String`
Read a line from the standard input device

type `FilePath = String`

readFile :: `FilePath -> IO String`
Returns the contents of the file as a string.

writeFile :: `FilePath -> String -> IO ()`
Writes a string to a file.

getArgs :: `IO [String]` Returns a list of the program's command line arguments (in `System.Environment`)

IO – A More Complex Example

- Read file name from the input, sort it, write it to the output

```haskell
import System.Environment (getArgs)

main = do
    args <- getArgs
    if null args
       then print "Provide a filename"
       else do
           fileCnt <- readFile (head args)
           let cnt :: [Int]
               cnt = map read (lines fileCnt)
           putStrLn (show (quickSort cnt))
           writeFile
               (mkName (head args))
               ("#sorted: " ++ show (length cnt))
  where
      mkName name = takeWhile (/= '.') name
          ++ ".out"
```

- For more detailed description of functions use *Hoogle*

Last lecture

- This was the last lecture
- Thank you for you patience
- Please send me a feedback or any comments to `frantisek@farka.eu`