Haskell: Notes

Moritz M. Konarski

October 9, 2020

Contents

Chapter 1

Variables and functions

Variables

- variables are useful

```haskell
a = 3.141592
a^3
```

- code is more readable and modifiable

Haskell source files

- file extension is `.hs`
- to load a file in GHCi type `:l`, to reload type `:r`

Comments

- comments are done like this

```haskell
x = 5        -- a comment
```

- block comments work like this

```haskell
x = 5
{-
    multi-line comment
-}
y = {- inline comment -} 12
```

Variables in imperative languages

- in Haskell, variables can only be declared once and they are immutable
- they must begin with a letter and then can contain letters, numbers, underscores and ticks

Functions

- functions take an argument (or parameter) and gives a resulting value
- they are defined as follows

```
area r = pi * r ^ 2
```

- functions don't use parentheses but they can be used to group expressions and to make code easier to read
- haskell functions can also take multiple arguments

```
areaTriangle b h = (b * h) / 2
```

- functions can also be passed as arguments
- arguments are applied in the order they are given
- functions can be used to defined some new functions

Local definitions

- when functions have values local to them, they are declared using `where` – see Heron's formula $A = \sqrt{s(s-a)(s-b)(s-c)}$

```
heron a b c = sqrt (s * (s - a) * (s - b) * (s - c))
    where
    s = (a + b + c) / 2
```

- the `where` and local variables are indented by 4 spaces – there can be multiple such statement

Chapter 2

Truth values

Equality and other comparisons

- double equal signs are used for comparisons `==`
- `True` and `False` are the representations of the booleans
- other evaluations are `<`, `>`, `>=`, `<=`, `/=`

```
-- defining operators
x /= y = not (x == y)
```

- types are important

Infix operators

- functions that are written between the arguments

```
4 + 9 == 13
-- same as
(==) (4 + 9) 13
```

Boolean operations

- logical and is `&&`
- logical or is `||`
- logical not is `not`

Guards

- syntactic sugar for piecewise functions

```
-- function for the absolute value of x
absolute x
    | x < 0     = 0 - x     -- -x would also work
    | otherwise = x
```

- the pipe `|` is followed by a predicate (boolean expression)

- `otherwise` is used when none of the preceding values are `True`, it is actually just defined as `True`
- `where` works well with guards

```
numOfRealSolutions a b c
    | disc > 0 = 2
    | disc == 0 = 1
    | otherwise = 0
      where
      disc = b^2 - 4*a*c      -- discriminant for above
                              -- descisions
```

Chapter 3

Type basics

- all types in haskell have to begin with a capital letter
- types are useful because they define what you can and can't do with them
- `:type` or `:t` checks the type of any expression
- `::` means 'is of type', it indicates the *type signature*
- `True` and `False` are of type `Bool`
- characters are of type char

```
:t 'H'
'H' :: Char
```

- strings are of type list of char

```
:t "hello"
"hello" :: [Char]
```

- type synonyms are different words for the same types

```
[Char] == String
```

Functional types

- functions have types too
- type signature for `not`

```
:t not
not :: Bool -> Bool
-- function from bool to bool
```

- `chr :: Int -> Char` converts int to char ASCII
- `ord :: Char -> Int` converts ASCII to int
- to use these functions you have to use the module `Data.Char` with `:module Data.Char` or `:m Data.Char`
- finding the type signatures of function works by listing all types of the input values in order and then the result value, all separated by `->`

```
xor p q = (p || q) && not (p && q)
:t xor
```

```haskell
xor :: Bool -> Bool -> Bool
```

Type signatures in code

- type annotations look exactly like the function signatures

```haskell
xor :: Bool -> Bool -> Bool
xor p q = (p || q) && not (p && q)
```

- this clarifies the function to the compiler and the programmer
- when types are not provided, the compiler infers the types by what data is there
- also, type signatures can help the compiler spot errors for you
- by separating functions with commas we can put multiple ones on the same signature line
- if one writes `+` instead of `++` for concatenation, the compiler will let you know

Chapter 4

Lists and tuples

Lists

- denoted by `[` and `]`, elements are separated by commas
- all elements must be of the same type

```
numbers = [1,2,3,4]
bools   = [True, False, False]
```

- to add elements to the start of a list, use `:` (cons), evaluated from right to left

```
[1,2,3,4] == 1:2:3:4:[]
```

- you can only cons elements to a list, not vice versa
- strings are also just lists of characters
- lists can also contain lists – a useful feature

Tuples

- store multiple values in a single value
- tuples will always have a set length, you cannot increase their size – good if you know the amoung of needed data
- elements of a tuple do not need to be of the same type

```
(True, 1)
("hello", 'c', 123.23)
```

- tuples can also contain other tuples
- `fst` and `snd` return the first and second elements of a 2-tuple or pair
- `head` and `tail` for lists return the first element and the list minus the first element
- head and tail are pretty bad though, they will fail if passed an empty list
- functions can use polymorphic types, they can represent a bunch of different types with certain similarities
- type variables allow any type to take their place, they are useful for writing functions that can work on many types
- mathematically that's called polymorphism

```haskell
f :: a -> a
-- takes type a and returns same type a
f :: a -> b
-- takes a and may or may not return the same type
```

- for example `fst` and `snd` work like this

```haskell
-- return first of pair
fst :: (a, b) -> a
-- return second of pair
snd :: (a, b) -> b
```

Chapter 5

Type basics II

- in maths you can add any type of number together – for computers that does not work too well
- floats and integers are the least types you need
- this means Haskell needs types for at least those two
- still, the `(+)` works on any type of number

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

- here `Num` is a typeclass – a restriction on the types that a function can accept
- the most important numeric types are `Int`, `Integer`, `Float`, and `Double`
- `Int` is 32 bit, `Integer` is arbitrarily long, the others are floating point numbers
- a number in a Haskell program is of type number and is only restricted when it is changed, like `7` is anything, `3.12` is restricted to double and then they are added on the lowest denominator
- monomorphic trouble comes when returned types are incompatible
- using a function that returns `Int` with a function expecting `Double` will blow up, it requires conversion

```
-- converts an int into a polymorphic number
fromIntegral(num)
```

Classes beyond numbers

- there is a typeclass for equations `Eq`
- length is a function that takes a `Foldable`, a type that includes lists and more

Chapter 6

Building vocabulary

Function composition

- means applying one function to a value and then applying another function to the result

```
-- defining functions
f x = x + 3
square x = x ^ 2
square (f 1)        -- returns 16
f (square 2)        -- returns 7
```

- the parentheses are necessary because otherwise the function would try to take another function as input – an error
- we can make one function of multiple commonly used ones

```
squareOfF x = square (f x)
fOfSquare x = f (square x)
```

- another way to do it is with `(.)`, the function composition operator

```
-- functions are applied from right to left
squareOfF x = (square . f) x
fOfsquare x = (f . square) x
```

- one can also leave out the `x`, giving

```
squareOfF = square . f
```

Prelude and the libraries

- the standard library in Haskell is called the prelude
- it provides the types and general functions
- you can import modules into your program

```
import Data.List
```

- in GHCi use `:m +<name>`

- using the standard library can make programs much shorter

```
-- program that reverses the order of strings
revWords :: String -> String
revWords input = (unwords . reverse . words) input
```

Chapter 7

Next steps

If then else

- works like in other languages

```
signum x =
    if x < 0
        then -1
        else if x > 0
            then 1
            else 0
```

- there *has* to be an `else` statement because there always has to be a result
- `if - then - else` can also be translated to guards

```
signu x =
    | x < 0     = -1
    | x > 0     =  1
    | otherwise =  0
```

- both options work the same, sometimes one is more readable than the other

Introducing pattern matching

- long `if-else` statements are convoluted
- defining piece-wise functions is much better and more readable

```
pts :: Int -> Int
pts 1 = 10
pts 2 = 6
pts 3 = 4
pts 4 = 3
pts 5 = 2
pts 6 = 1
pts _ = 0
```

- the _ is a wildcard – matching anything of the functions type

- using some logic, we can make `pts` even shorter

```
pts :: Int -> Int
pts 1 = 10
pts 2 = 6
pts x
    | s <= 6     = 7 - x
    | otherwise = 0
```

- pattern matching can be overlapped and thus inefficient
- patterns should always match all possible cases

Tuple and list patterns

- the patterns work the same, they are more useful actually

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

tail         :: [a] -> [a]
tail (_:xs) = xs
tail []      = error "Prelude.head: empty list"
```

`Let` bindings

- similar to `where`, different formatting
- can help make functions more readable

```
roots a b c =
    ((-b + sqrt(b * b - 4 * a * c)) / (2 * a),
     (-b - sqrt(b * b - 4 * a * c)) / (2 * a))
-- can be transformed to
short_roots a b c =
    let d = sqrt (b ^ 2 - 4 * a * c)
        two_a = 2 * a
    in ((-b + d) / two_a,
        (-b - d) / two_a)
```

Chapter 8

Simple input and output

- program have to interact with the world around them

```
putStrLn "Hello, world!"
```

- `putStr` just misses the new line `putStrLn` has
- these functions are of type `IO ()`, that is `IO` action with type `()` – unit type, tuple with 0 elements – basically nothing
- `IO` occurs in print, read, write

Sequencing actions with `do`

- `do` lets you put actions in order of execution

```
main = do
    putStrLn "Please enter your name:"
    name <- getLine
    putStrLn ("Hello, " ++ name ++ ", how are you")
```

- the final action defines the type of the whole `do` block – here the whole program has type `IO ()`
- the `<-` notation assigns the result of `getLine` to the variable – omitting it will just take the input but don't act on it
- `<-` cannot be the last action
- use `read` to convert string to text, show to convert number to string

```
double = read "123.3"

string_of_double = show double
```

- `<-` can be used to get any result value, just some of them are not worth saving
- the two branches of if/then/else statements must all have the same type
- for all sequenced commands we need to start a new `do` block
- for action to return what they say they will, you require the `<-`

Chapter 9

Recursion

- a function that has the ability to invoke itself
- a recursive function calls itself in certain circumstances and stops in others

Numeric recursion

- see factorials

```
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

- always write the most restrictive type first
- `go` can be used to translate loops from other languages into Haskell
- many functions have recursive versions – multiplication for example

List-based recursion

- lists make extensive use of recursion, e.g. for length

```
length :: [a] -> Int
length []       = 0
length (x:xs)   = 1 + length xs
```

- recursion is the only way to implement control structures if one only has immutable types
- the list concatenate function is defined recursively too

```
(++) :: [a] -> [a] -> [a]
[] ++ ys        = ys
(x:xs) ++ ys    = x : xs ++ ys
-- basically turns [1,2,3] ++ [4,5,6] into 1:2:3:[4,5,6]
```

- list recursion generally involves an empty list case and then one where the tail is passed to the function again

Chapter 10

Lists II

Rebuilding lists

- function that doubles all elements in a list

```
doubleList :: [Integer] -> [Integer]
doubleList [] = []
doubleList (n:ns) = (2 * n) : doubleList ns
```

- to create a multiply list function, we add another factor

```
multiplyList :: [Integer] -> [Integer]
multiplyList _ [] = []
multiplyList m (n:ns) = (m * n) : multiplyList m ns
```