Recollection Haskell, Part II
The Basics of Types and Function Syntax

(Based on Chapters 2 and 3 of LYH)

CIS 352: Programming Languages

January 17, 2019

Types

> ***Well-typed programs cannot "go wrong."*** — Robin Milner

- Evaluation preserves well-typedness.
- A well-typed program never gets stuck (in an undefined state).

Well-typedness is a *safety property*.

Safety	$\equiv$	some particular bad thing never happens.
Liveness	$\equiv$	some particular good thing eventually happens.

Haskell is a *strongly typed* language.

- strongly typed $\Longrightarrow$ good safety properties
- strongly typed $\Longrightarrow$ *very* fussy
- weakly typed $\Longrightarrow$ you can get away with murder
  but often you are the victim

Arguing with the type inference engine *(it always wins)*

```
ghci> ['a',('q','z')]

<interactive>:1:6: error:
    Couldn't match expected type ‘Char’ with actual type ‘(t0, t1)’
    In the expression: ('q', 'z')
    In the expression: ['a', ('q', 'z')]
    In an equation for ‘it’: it = ['a', ('q', 'z')]

ghci> :t 'a'
'a' :: Char
ghci> :t ('q','z')
('q','z') :: (Char, Char)

ghci> :t 4==5
4==5 :: Bool
```

Explicitly declaring types

someFuns.hs

```
zapUpper :: [Char] -> [Char]
zapUpper cs = [ c | c <- cs, c `notElem` ['A'..'Z']]

addThree :: Int -> Int -> Int -> Int
addThree x y z = x+y+z
```

Standard types

- `Int`
- `Integer`
- `Float`
- `Double`
- `Bool`
- `Char`
- Type variables: `a`, `b`, `c`, `x`. . . , `t`, `t1`, `t2`, . . .
- Tuple types: `(), (t1,t2), (t1,t2,t3), ...`
- `List types: [t]`

Type classes, a first look

Type classes are “clubs” types can join.

There are:

- membership requirements,
- membership benefits, and
- membership cards you can show to get into places

Some standard type classes:

`http://haskell.org/onlinereport/basic.html#sect6.3`

The Eq type class

```
ghci> :i Eq
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  -# MINIMAL (==) | (/=) #-
   -- Defined in ‘GHC.Classes’
instance Eq a => Eq [a] -- Defined in ‘GHC.Classes’
      ⋮
ghci> :t (==)
(==) :: Eq a => a -> a -> Bool
```

sample.hs

```
twoEqChar :: Char -> Char -> Char -> Bool
twoEqChar c1 c2 c3 =  (c1==c2) || (c1==c3) || (c2==c3)

twoEq :: (Eq a) => a -> a -> a -> Bool
twoEq x1 x2 x3 =  (x1==x2) || (x1==x3) || (x2==x3)
```

Some other type classes

Ord — for types that can be put in an order

```
ghci> :t (<)
(<) :: Ord a => a -> a -> Bool
```

Show — for types that can be printed
Read — for types that can be read
Enum — for sequentially ordered types
Bounded — for types with lower and upper bounds
Num — for numeric types
Floating — for floating point types
Integral — for whole number types

```
ghci> fromIntegral (length [1,2,3,4]) + 3.2
7.2
```

Defining functions

Haskell program	$\approx$	series of definitions and comments
Haskell definition	$\approx$	type declarations + equations

General format

$$\texttt{name :: } \underbrace{\texttt{t1 -> t2 -> ... -> tk}}_{\textit{argument types}} \texttt{ -> } \underbrace{\texttt{t}}_{\textit{result type}}$$

```
name x1 x2 ... xk = e
```

variables `x1 :: t1, x2 :: t2, ... , xk :: tk`

expression `e :: t`

Examples

```
isPositive :: Int -> Bool
isPositive num = (num>0)

foo :: Int -> Int -> Int
foo x y = x + (twice y) - 6
```

Patterns: Constants and Variables

- A function definition can be a sequence of equations.
- When a function is applied to some values, the equations are tried from top to bottom to find one that "succeeds" for these values.
- The form of the left-hand-side of a defining equation is

  `funName` $pat_1 \; \ldots \; pat_n$

- A pattern that is a constant value matches only that value.
- A pattern that is a variable matches any value.

```
lucky7 :: Int -> String
lucky7 7 = "You win"
lucky7 x = "You loose"
```

```
myFun :: Int -> Int -> Int
myFun 0 y = 15
myFun x 0 = x + 11
myFun x y = x + y * y + 3
```

What happens if none of the equations succeed?

Patterns: Tuples

Correct, …

```
addVectors :: (Double, Double) -> (Double, Double) -> (Double, Double)
addVectors a b = (fst a + fst b, snd a + snd b)
```

Patterns: Tuples

Correct, . . .

```
addVectors :: (Double, Double) -> (Double, Double) -> (Double, Double)
addVectors a b = (fst a + fst b, snd a + snd b)
```

but this is preferred

```
addVectors' :: (Double, Double) -> (Double, Double) -> (Double, Double)
addVectors' (x1, y1) (x2, y2) = (x1 + x2, y1 + y2)
```

Patterns: Tuples

Correct, . . .

```
addVectors :: (Double, Double) -> (Double, Double) -> (Double, Double)
addVectors a b = (fst a + fst b, snd a + snd b)
```

but this is preferred

```
addVectors' :: (Double, Double) -> (Double, Double) -> (Double, Double)
addVectors' (x1, y1) (x2, y2) = (x1 + x2, y1 + y2)
```

```
first  :: (a, b, c) -> a
first (x, _, _) = x

second :: (a, b, c) -> b
second (_, y, _) = y

third  :: (a, b, c) -> c
third (_, _, z) = z
```

_ is the wildcard pattern—it matches anything.

Patterns: List comprehensions

```
ghci> [a+b | (a,b) <- [(2,3),(9,4),(0,5)]]
[5,13,5]

ghci> [a | (a,5) <- [(10,5),(2,3),(9,4),(0,5)]]
[10,0]
```

Patterns: Lists

The (x:xs) pattern

```
head' :: [a] -> a
head' (x:_) = x
head' []    = error "Can't call head on an empty list, silly!"
```

```
tell :: (Show a) => [a] -> String
tell []         = "The list is empty"
tell (x:[])     = "The list has one element: " ++ show x
tell (x:y:[])   = "The list has two elements: " ++ show x
                    ++ " and " ++ show y
tell (x:y:_)    = "This list is long. The first two elements are: "
                    ++ show x ++ " and " ++ show y
```

```
okAdd, betterAdd :: (Num a) => [a] -> a
okAdd (x:y:z:_) = x + y + z

betterAdd xs = sum (take 3 xs)                                    (Why better?)
```

Guards: Often patterns are not enough to distinguish cases

```
name p1 ... pk
     | test1      = e1
     | test2      = e2
         ...
     | otherwise  = ek
```

$\cong$

```
name p1 ... pk =
     if        test1 then e1
     else if test2 then e2
         ...
     else                 ek
```

Examples

```
-- This is a redefinition of the max function.
max :: Int -> Int -> Int
max x y
    | (x<=y)     = y
    | otherwise = x
-- maxThree x y z = the max of the three numbers
maxThree :: Int -> Int -> Int -> Int
maxThree x y z
    | (x>=y) && (x>=z) = x
    | (y>=z)           = y
    | otherwise        = z
```

Aside: Guards, more generally

If you have a function definition that includes the line

```
   ⋮
fun p1 ... pk | test = e
   ⋮
```

It means that:

if the patterns match *and* `test` succeeds
  then return the value of `e`
  else don't use this line but try the next.

E.g.:

```
fact n | n<0 = error "fact given a negative argument"
fact 0       = 1
fact n       = n * fact (n-1)
```

where and let, 1

You can introduce ***local variables*** that are visible only inside a definition. E.g.

```
maxSq :: Int -> Int -> Int
maxSq x y = max x2 y2
    where
      x2 = x*x  -- x2 is a local variable to maxSq
      y2 = y*y  -- y2 is a local variable to maxSq

maxSq' :: Int -> Int -> Int
maxSq' x y = max (sq x) (sq y)
    where
      sq x = x * x  -- sq is a function def local to maxSq'
```

Alternatively,

```
maxSq'' :: Int -> Int -> Int
maxSq'' x y =
    let sq x = x * x  -- sq is a function def local to maxSq'
    in max (sq x) (sq y)
```

- `let`'s are expressions
- `where` is part of the syntax for function definitions

✔ `let y = (let x = 3 in x+2) in y+11`

✘ `let y = (x+2 where x=3) in y+11`

✔
```
f x y  |  y>z        =  ...
       |  y==z       =  ...
       |  y<z        =  ...
     where z = x*x
```

✘
```
let z = x*x
 in f x y  |  y>z        =  ...
           |  y==z       =  ...
           |  y<z        =  ...
```

Case statements: Pattern matching in expressions

Syntax

```
case expression of
      pattern1 -> result1
      pattern2 -> result2
      pattern3 -> result3
      ...
```

```
describeList, describeList' :: [a] -> String
describeList ls
    = "The list is " ++ case ls of []  -> "empty."
                                   [x] -> "a singleton list."
                                   xs  -> "a longer list."
```

Case statements: Pattern matching in expressions

Syntax

```
case expression of
      pattern1 -> result1
      pattern2 -> result2
      pattern3 -> result3
      ...
```

```
describeList, describeList' :: [a] -> String
describeList ls
    = "The list is " ++ case ls of []  -> "empty."
                                   [x] -> "a singleton list."
                                   xs  -> "a longer list."
```

```
-- alternatively
describeList' ls = "The list is " ++ what ls
    where what []  = "empty."
          what [x] = "a singleton list."
          what xs  = "a longer list."
```

Layout: Indentation matters!!!

- Haskell has a 2D syntax*.
- Basic idea: layout determines where a definition start & stops
- The Rule: A definition ends at the first piece of text that lies at the same indentation (or to the left of) the start of that definition.

✔
```
-- OK, if ugly.
fun1 :: Int -> Int
fun1 x = x
  +1
```

✘
```
-- This is misformated
fun2 :: Int -> Int
fun2 x = x
+1
```

✘
```
-- This is also bad
fun3, fun4 :: Int -> Int
fun3 x = x + 10
  fun4 x = x *20
```

* But there are {'s, }'s, and ;'s around if you really, *really* need them.