Recollection Haskell, Part II
The Basics of Types and Function Syntax
(Based on Chapters 2 and 3 of LYHGG)

CIS 352/Spring 2015
Programming Languages

January 15, 2015

Types

Well-typed programs never 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.

<table>
<thead>
<tr>
<th>Safety</th>
<th>Liveness</th>
</tr>
</thead>
<tbody>
<tr>
<td>≡ some particular bad thing never happens.</td>
<td>≡ some particular good thing eventually happens.</td>
</tr>
</tbody>
</table>

Haskell is a strongly typed language.

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

Arguing with the type inference engine  (it always wins)

```
*Main> ['a',('q','z')]
<interactive>:1:6:
  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')]

*Main> :t 'a'
'a' :: Char

*Main> :t ('q','z')
('q','z') :: (Char, Char)

*Main> :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
```

CIS 352 (Programming Languages) Types and Functions January 15, 2015 1 / 18

CIS 352 (Programming Languages) Types and Functions January 15, 2015 2 / 18

CIS 352 (Programming Languages) Types and Functions January 15, 2015 3 / 18

CIS 352 (Programming Languages) Types and Functions January 15, 2015 4 / 18
### 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

```haskell
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
      -- Defined in GHC.Classes
instance Eq a => Eq (Maybe a) -- Defined in Data.Maybe
instance Eq Int -- Defined in GHC.Base

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

```haskell
(>) :: 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
```

```haskell
fromIntegral (length [1,2,3,4]) + 3.2
7.2
```
Defining functions

Haskell program ≈ series of definitions and comments
Haskell definition ≈ type declarations + equations

General format

```
name :: t1 -> t2 -> ... -> tk -> t
```

- **argument types**
- **result type**

- **variables**
- **expression**

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 ... 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)
```

but this is preferred

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

Correct...

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

but this is preferred

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

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

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

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

_ is the wildcard pattern—it matches anything.

Patterns: List comprehensions

```haskell
*Main> [a+b | (a,b) <- [(2,3),(9,4),(0,5)]]
[5,13,5]
*Main> [a | (a,5) <- [(10,5),(2,3),(9,4),(0,5)]]
[10,0]
```

Patterns: Lists

The `(x:xs)` pattern

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

```haskell
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:z:) = "This list is long. The first two elements are: " ++ show x ++ " and " ++ show y
```

```haskell
badAdd, betterAdd :: (Num a) => [a] -> a
badAdd (x:y:z:[]) = x + y + z
betterAdd xs = sum (take 3 xs)
```

(Why better?)

Guards

Often patterns are not enough to distinguish cases

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

Examples

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

```haskell
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
```

Alternatively,

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

How are these two things different?

- *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`

```haskell
let z = x*x

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

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

Case statements: Pattern matching in expressions

**Syntax**

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

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

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

--- alternatively

```haskell
describeList' ls = "The list is " ++ what ls
    where what [] = "empty."
        what [x] = "a singleton list."
        what xs = "a longer list."
```
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 '{', '}', and ';'s around if you really, really need them.