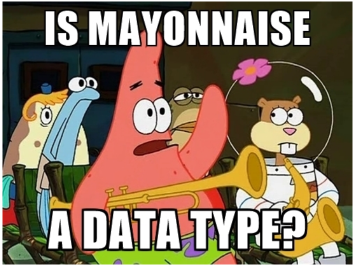

Types and Pattern Matching

Stephen A. Edwards

Columbia University

Basic Haskell Types

Function Types

Patterns

- The As Pattern
- Guards

Algebraic Data Types

Types in Haskell

Haskell is statically typed: every expression's type known at compile-time

Haskell has type inference: the compiler can deduce most types itself

Type names start with a capital letter (Int, Bool, Char, etc.)

GHCi's `:t` command reports the type of any expression

Read "`::`" as "is of type"

```
Prelude> :t 'a'
'a' :: Char

Prelude> :t True
True :: Bool

Prelude> :t "Hello"
"Hello" :: [Char]

Prelude> :t (True, 'a')
(True, 'a') :: (Bool, Char)

Prelude> :t 42 == 17
42 == 17 :: Bool
```

Some Common Types

`Bool`	Booleans: `True` or `False`
`Char`	A single Unicode character, about 25 bits
`Int`	Word-sized integers; the usual integer type. E.g., 64 bits on my x86_64 Linux desktop
`Integer`	Unbounded integers. Less efficient, so only use if you need *really* big integers
`Float`	Single-precision floating point
`Double`	Double-precision floating point

The Types of Functions

In a type, -> indicates a function

```
Prelude> welcome x = "Hello " ++ x
Prelude> welcome "Stephen"
"Hello Stephen"
Prelude> :t welcome
welcome :: [Char] -> [Char]
```

“Welcome is a function that takes a list of characters and produces a list of characters”

Multi-argument functions are Curried

Haskell functions have exactly one argument.
Functions with "multiple arguments" are actually functions that return functions that return functions.

Such "currying" is named after Haskell Brooks Curry, who is also known for the Curry-Howard Correspondence ("programs are proofs").

```
Prelude> say x y = x++" to "++y
Prelude> :t say
say :: [Char] -> [Char] -> [Char]
Prelude> say "Hello" "Stephen"
"Hello to Stephen"

Prelude> :t say "Hello"
say "Hello" :: [Char] -> [Char]
```

```
Prelude> hello s = say "Hello" s
Prelude> hello "Fred"
"Hello to Fred"
Prelude> :t hello
hello :: [Char] -> [Char]
Prelude> hello = say "Hello"
Prelude> hello "George"
"Hello to George"
Prelude> :t hello
hello :: [Char] -> [Char]
```

Top-level Type Declarations

It is good style in `.hs` files to include type declarations for top-level functions

Best documentation ever: a precise, compiler-verified function summary

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

```
Prelude> :l addThree
[1 of 1] Compiling Main             ( addThree.hs, interpreted )
Ok, one module loaded.
*Main> :t addThree
addThree :: Int -> Int -> Int -> Int
*Main> addThree 1 2 3
6
```

Patterns

You can define a function with patterns

Patterns may include literals, variables, and _ “wildcard”

```
badCount :: Integral a => a -> String
badCount 1 = "One"
badCount 2 = "Two"
badCount _ = "Many"
```

Patterns are tested in order; put specific first:

```
factorial :: (Eq a, Num a) => a -> a
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Pattern Matching May Fail

```
Prelude> :{
Prelude| foo 'a' = "Alpha"
Prelude| foo 'b' = "Bravo"
Prelude| foo 'c' = "Charlie"
Prelude| :}
Prelude> :t foo
foo :: Char -> [Char]
Prelude> foo 'a'
"Alpha"
Prelude> foo 'd'
"*** Exception: <interactive>:(23,1)-(25,19): Non-exhaustive
                                           patterns in function foo
```

Let the Compiler Check for Missing Cases

Much better to get a compile-time error than a runtime error:

```
Prelude> :set -Wall
Prelude> :{
Prelude| foo 'a' = "Alpha"
Prelude| foo 'b' = "Bravo"
Prelude| :}

<interactive>:32:1: warning: [-Wincomplete-patterns]
    Pattern match(es) are non-exhaustive
    In an equation for 'foo':
        Patterns not matched: p where p is not one of {'b', 'a'}
```

```
Prelude> :set -Wincomplete-patterns
```

Pattern Matching on Tuples

A tuple in a pattern lets you dismantle the tuple. E.g., to implement *fst*,

```
Prelude> fst' (x,_) = x
Prelude> :t fst'
fst' :: (a, b) -> a
Prelude> fst' (42,28)
42
Prelude> fst' ("hello",42)
"hello"
```

```
Prelude> addv (x1,y1) (x2,y2) = (x1 + x2, y1 + y2)
Prelude> :t addv
addv :: (Num a, Num b) => (a, b) -> (a, b) -> (a, b)
Prelude> addv (1,10) (7,3)
(8,13)
```

Patterns in List Comprehensions

Usually, where you can bind a name, you can use a pattern, e.g., in a list comprehension:

```
Prelude> :set +m
Prelude> pts = [ (a,b,c) | c <- [1..20], b <- [1..c], a <- [1..b],
Prelude|                   a^2 + b^2 == c^2 ]
Prelude> pts
[(3,4,5),(6,8,10),(5,12,13),(9,12,15),(8,15,17),(12,16,20)]

Prelude> perimeters = [ a + b + c | (a,b,c) <- pts ]

Prelude> perimeters
[12,24,30,36,40,48]
```

Pattern Matching On Lists

You can use `:` and `[,,,]`-style expressions in patterns

Like *fst*, *head* is implemented with pattern-matching

```
Prelude> :{
Prelude| head' (x:_) = x
Prelude| head' [] = error "empty list"
Prelude| :}

Prelude> :t head'
head' :: [p] -> p

Prelude> head' "Hello"
'H'
```

Pattern Matching On Lists

```
Prelude> :{
Prelude| dumbLength [] = "empty"
Prelude| dumbLength [_] = "singleton"
Prelude| dumbLength [_,_] = "pair"
Prelude| dumbLength [_,_,_] = "triple"
Prelude| dumbLength _ = "four or more"
Prelude| :}

Prelude> :t dumbLength
dumbLength :: [a] -> [Char]
Prelude> dumbLength []
"empty"
Prelude> dumbLength [1,2,3]
"triple"
Prelude> dumbLength (replicate 10 ' ')
"four or more"
```

List Pattern Matching Is Useful on Strings

```
Prelude> :{
Prelude| notin ('i':'n':xs) = xs
Prelude| notin xs = "in" ++ xs
Prelude| :}

Prelude> notin "inconceivable!"
"conceivable!"
Prelude> notin "credible"
"incredible"
```

Pattern Matching On Lists with Recursion

```
Prelude> :{
Prelude| length' [] = 0
Prelude| length' (_:xs) = 1 + length' xs
Prelude| :}
Prelude> :t length'
length' :: Num p => [a] -> p
Prelude> length' "Hello"
5
```

```
Prelude> :{
Prelude| sum' [] = 0
Prelude| sum' (x:xs) = x + sum' xs
Prelude| :}
Prelude> sum' [1,20,300,4000]
4321
```

The “As Pattern” Names Bigger Parts

Syntax: <name>@<pattern>

```
Prelude> :{
Prelude| initial "" = "Nothing"
Prelude| initial all@(x:_) = "The first letter of " ++ all ++
Prelude|                     " is " ++ [x]
Prelude| :}

Prelude> :t initial
initial :: [Char] -> [Char]
Prelude> initial ""
"Nothing"
Prelude> initial "Stephen"
"The first letter of Stephen is S"
```

Guards: Boolean constraints

Patterns match structure; guards (Boolean expressions after a |) match value

```
Prelude> :{
Prelude| heightEval h
Prelude|   | h < 150 = "You're short"
Prelude|   | h < 180 = "You're average"
Prelude|   | otherwise = "You're tall"     -- otherwise = True
Prelude| :}

Prelude> heightEval 149
"You're short"
Prelude> heightEval 150
"You're average"
Prelude> heightEval 180
"You're tall"
```

Filter: Keep List Elements That Satisfy a Predicate

odd and *filter* are Standard Prelude functions

```
odd n = n `rem` 2 == 1

filter :: (a -> Bool) -> [a] -> [a]
filter p []                     = []
filter p (x:xs) | p x           = x : filter p xs
                | otherwise = filter p xs
```

```
Prelude> filter odd [1..10]
[1,3,5,7,9]
```

Compare: Returns LT, EQ, or GT

Another Standard Prelude function

```
x `compare` y
  | x < y     = LT
  | x == y    = EQ
  | otherwise = GT
```

```
Prelude> :t compare
compare :: Ord a => a -> a -> Ordering
Prelude> compare 5 3
GT
Prelude> compare 5 5
EQ
Prelude> compare 5 7
LT
Prelude> 41 `compare` 42
LT
```

Where: Defining Local Names

```
triangle :: Int -> Int -> Int -> String
triangle a b c
   | a + b < c  || b + c < a  || a + c < b  = "Impossible"
   | a + b == c || a + c == b || b + c == a = "Flat"
   | right                                  = "Right"
   | acute                                  = "Acute"
   | otherwise                              = "Obtuse"
   where
     right = aa + bb == cc || aa + cc == bb || bb + cc == aa
     acute = aa + bb > cc  && aa + cc > bb  && bb + cc > aa
     sqr x = x * x
     (aa, bb, cc) = (sqr a, sqr b, sqr c)
```

Order of the *where* clauses does not matter

Indentation of the *where* clauses must be consistent

Where blocks are attached to declarations

The Primes Example

```
primes = filterPrime [2..]
  where filterPrime (p:xs) =
          p : filterPrime [x | x <- xs, x `mod` p /= 0]
```

`[2..]`	The infinite list [2,3,4,...]
where `filterPrime`	*Where* clause defining *filterPrime*
`(p:xs)`	Pattern matching on head and tail of list
`p : filterPrime ...`	Recursive function application
`[x | x <- xs, x ‘mod‘ p /= 0]`	List comprehension: everything in xs not divisible by `p`

`case...of` Is a Pattern-Matching Expression

Defining a function with patterns is syntactic sugar for *case...of*

```
badCount 1 = "One"
badCount 2 = "Two"
badCount _ = "Many"
```

is equivalent to

```
badCount x = case x of
    1 -> "One"
    2 -> "Two"
    _ -> "Many"
```

But, like *let*, *case...of* is an expression and may be used as such:

```
describeList :: [a] -> String
describeList xs = "The list is " ++ case xs of []  -> "empty"
                                               [x] -> "a singleton"
                                               _   -> "two or more"
```

Algebraic Data Types

```
data Bool = False | True
```

`Bool`: Type Constructor `False` and `True`: Data Constructors

```
Prelude> data MyBool = MyFalse | MyTrue

Prelude> :t MyFalse
MyFalse :: MyBool      -- A literal
Prelude> :t MyTrue
MyTrue :: MyBool

Prelude> :t MyBool
<interactive>:1:1: error: Data constructor not in scope: MyBool
Prelude> :k MyBool
MyBool :: *           -- A concrete type (no parameters)
```

Algebraic Types and Pattern Matching

```
data Bool = False | True
```

Type constructors may appear in type signatures;
data constructors in expressions and patterns

```
Prelude> :{
Prelude| myAnd :: Bool -> Bool -> Bool
Prelude| myAnd False _ = False
Prelude| myAnd True  x = x
Prelude| :}

Prelude> [ (a,b,myAnd a b) | a <- [False, True], b <- [False, True] ]
[(False,False,False),(False,True,False),
 (True,False,False),(True,True,True)]
```

An Algebraic Type: A Sum of Products

```
data Shape = Circle Float Float Float
           | Rectangle Float Float Float Float
```

Sum = one of A or B or C...

Product = each of D and E and F...

A.k.a. tagged unions, sum-product types

Mathematically,

$Shape = Circle \cup Rectangle$

$Circle = Float \times Float \times Float$

$Rectangle = Float \times Float \times Float \times Float$

An Algebraic Type: A Sum of Products

```haskell
data Shape = Circle Float Float Float
           | Rectangle Float Float Float Float

area                         :: Shape -> Float
area (Circle _ _ r)          = pi * r ^ 2
area (Rectangle x1 y1 x2 y2) = (abs $ x2 - x1) * (abs $ y2 - y1)
```

```
*Main> :t Circle
Circle :: Float -> Float -> Float -> Shape
*Main> :t Rectangle
Rectangle :: Float -> Float -> Float -> Float -> Shape
*Main> :k Shape
Shape :: *

*Main> area $ Circle 10 20 10
314.15927
*Main> area $ Rectangle 10 10 20 30
200.0
```

Types as Documentation

When in doubt, add another type

```haskell
data Point = Point Float Float deriving Show
data Shape = Circle Point Float
           | Rectangle Point Point
           deriving Show

area :: Shape -> Float
area (Circle _ r) = pi * r ^ 2
area (Rectangle (Point x1 y1) (Point x2 y2)) =
    (abs $ x2 - x1) * (abs $ y2 - y1)
```

```
*Main> area $ Rectangle (Point 10 20) (Point 30 40)
400.0
*Main> area $ Circle (Point 0 0) 100
31415.928
```

```haskell
moveTo :: Point -> Shape -> Shape
moveTo p (Circle _ r) = Circle p r
moveTo p@(Point x0 y0) (Rectangle (Point x1 y1) (Point x2 y2)) =
    Rectangle p $ Point (x0 + x2 - x1) (y0 + y2 - y1)

origin :: Point
origin = Point 0 0

originCircle :: Float -> Shape
originCircle = Circle origin -- function in "point-free style"

originRect :: Float -> Float -> Shape
originRect x y = Rectangle origin (Point x y)
```

```
Prelude> :l Shapes
[1 of 1] Compiling Shapes           ( Shapes.hs, interpreted )
Ok, one module loaded.
*Shapes> moveTo (Point 10 20) $ originCircle 5
Circle (Point 10.0 20.0) 5.0
*Shapes> moveTo (Point 10 20) $ Rectangle (Point 5 15) (Point 25 35)
Rectangle (Point 10.0 20.0) (Point 30.0 40.0)
```