Haskell

Data Types

The most advanced type system you've likely seen.

Very powerful

Uses Type Inferencing

Has a hierarchy of types: Type Classes

Primitive Types

Type	"Literals"	Info
`Bool`	`True` `False`	
`Int`	2	At least [-2
`Integer`	2	Arbitrary precision
`Float`	2	IEEE single precision

Type	"Literals"	Info
`Double`	2	IEEE double precision
Char	'a'	Enumeration of Unicode (ISO/IEC 10646)

Constructed Types

Type	Type Constuctors	Info
`Rational`	`4 % 5`	Arbitrary precision rational numbers. In Data.Ratio module

`Ordering`	`LT`
	`EQ`
	`GT`	

`String`	`['a'..'z']`	List of Char

Constructed Types

Type	Type Constuctors	Info
`Maybe`	`Nothing` `Just a`	Nothing or Just something
`Either`	`Left a` `Right b`	one or the other
…		Anything else you want to make

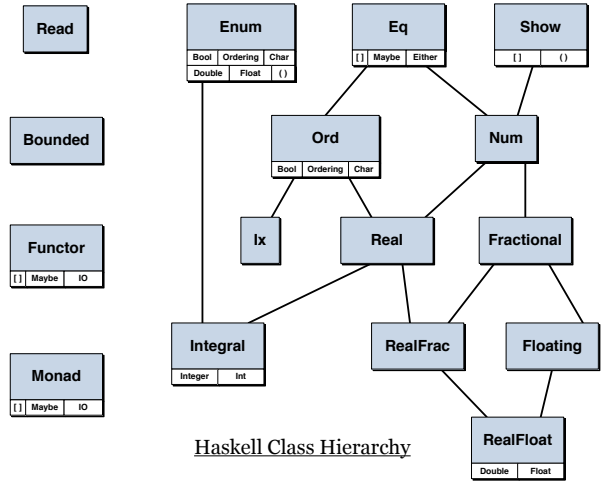
Composite Types

Type	Type Constuctors	Info

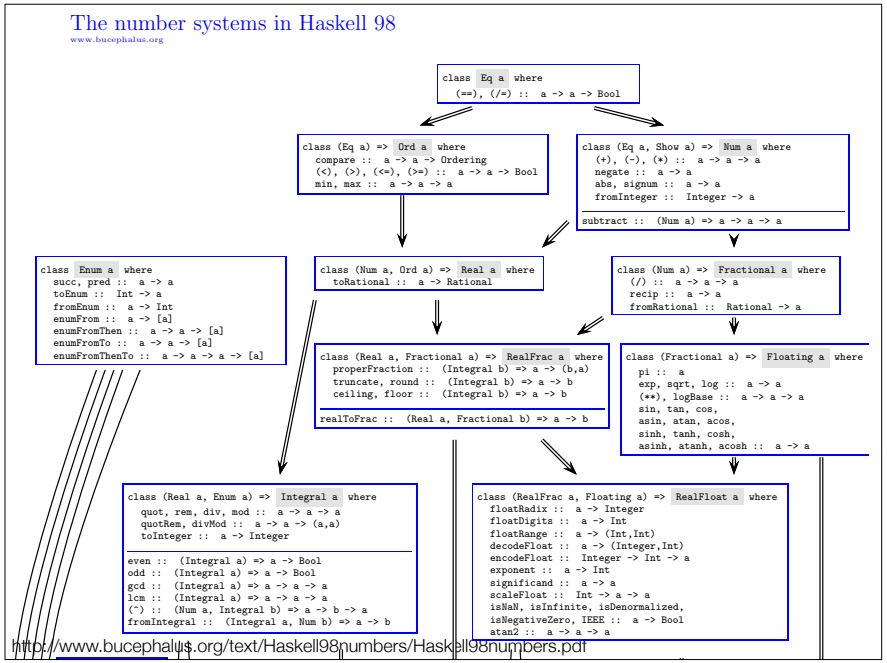
`[]`	`[]`
	`[1,2,3]`
	`[9..28]`
	`[1..]`
		Lists, must be homogeneous

`()`	`()` `('a',5,[1..])`	Tuple, can be heterogeneous

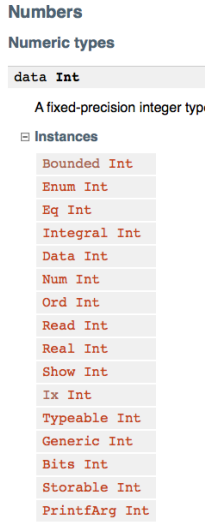
Type Classes Define Behavior



http://blogs.msdn.com/b/saeed/archive/2009/03/14/haskell-class-hierarchy-diagram.aspx



data Int



Guess the Type?

```
54
54.9
True
4 < 9
'a'
"Hello World"
(1,'a')
[4.5,9]
```

```
\x -> 2*x
```

In Haskell, typing is

Very strongly typed

Try adding an Int and a Double

Static: All type checking done at compile time

Optional: Types can be inferred

If you don't — you get the most general types

If you do specify it, then you get precisely that

Syntax

Syntax NOT similar to C based languages

Whitespace is important (like Python)

No parenthesis needed, but they're nice sometimes

No curly braces

Understanding precedence is important

Control Flow

None (in the traditional sense)

Only the rules of precedence for operators and functions

Function calls

Pattern matching in function definitions

if then else expression

case expression

No loops — only recursion

I/O looks different (procedural like), but isn't

Execution is lazy (you can write infinite data structures)

No variables; no persisting state information

NO side effects

Haskell is a pure functional language — what did you expect? You use functions to solve problems.

Functions

Function application

```
functionName  arg1  arg2  arg3
```

Function definition

```
functionName :: type1 -> type2 -> type3 -> resultType
functionName x y z = body
```

Typical simple function

```haskell
tempF :: Double -> Double   -- Convert Celsius to Fahrenheit
tempF c = 9 * c / 5 + 32
```

With a where clause

```haskell
mpg :: Double -> Double     -- Convert l/100km to miles per gallon
mpg fc = 100 * alpha / (fc * beta )
    where
        alpha = 3.78541     -- liters / gallon
        beta  = 1.60934     -- km / mile
```

With multiple definitions, and as an infix operator

```haskell
xor :: Bool -> Bool -> Bool
True `xor` True   = False
True `xor` False  = True
False `xor` True  = True
False `xor` False = False
```

With guards

```haskell
feelsLike :: Double -> String
feelsLike t
    | t < 0.0       = "Pretty damn cold"
    | t < 32.0      = "Freezing"
    | t < 40        = "Cold"
    | t < 60        = "Cool"
    | t < 80        = "Nice"
    | t < 100       = "Hot"
    | otherwise     = "Fracking Hot!"
```

With pattern matching

```haskell
reverseFirstThree :: [a] -> [a]
reverseFirstThree []        = []
reverseFirstThree [x]       = [x]
reverseFirstThree (x:y:[])  = [y,x]
reverseFirstThree (x:y:z:zs) = z:y:x:zs
```

More slides to follow:

- recursion
- lambda functions
- Higher order functions, currying
- first-class functions
- folds