Programming Language Concepts: Lecture 14

Madhavan Mukund

Chennai Mathematical Institute

madhavan@cmi.ac.in

http://www.cmi.ac.in/~madhavan/courses/pl2009

PLC 2009, Lecture 14, 11 March 2009

Function programming

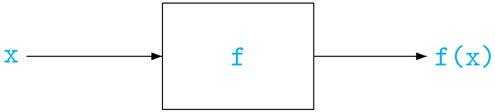
- A quick review of Haskell
- The (untyped) λ-calculus
- Polymorphic typed λ-calculus and type inference

Haskell

- Strongly typed functional programming language

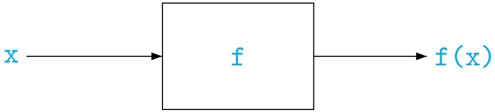
Haskell

- Strongly typed functional programming language
- Functions transform inputs to outputs:



Haskell

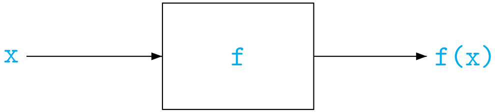
- Strongly typed functional programming language
- Functions transform inputs to outputs:



- A Haskell program consists of rules to produce an output from an input

Haskell

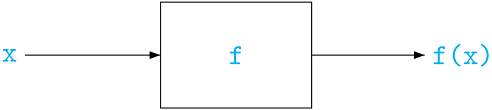
- Strongly typed functional programming language
- Functions transform inputs to outputs:



- A Haskell program consists of rules to produce an output from an input
- Computation is the process of applying the rules described by a program

Defining functions

A function is a black box:



Internal description of function `f` has two parts:

1. Types of inputs and outputs
2. Rule for computing the output from the input

Example:

`sqr :: Int -> Int`	Type definition	$sqr : \mathbb{Z} \to \mathbb{Z}$
`sqr x = x^2`	Computation rule	$x \mapsto x^2$

Basic types in Haskell

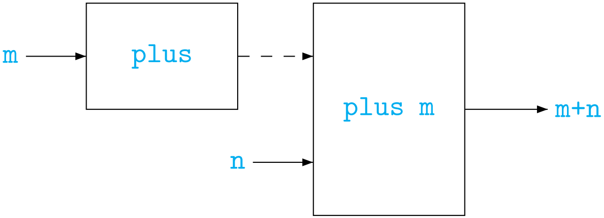
- `Int` Integers
  - Operations `+`, `-`, `*`
  - Functions `div`, `mod`
  - Note: `/ :: Int -> Int -> Float`
- `Float`
- `Char`
  - Values written in single quotes — `'z'`, `'&'`, ...
- `Bool`
  - Values `True` and `False`.
  - Operations `&&`, `||`, `not`

Functions with multiple inputs

- $plus(m, n) = m + n$
  - $plus : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$, or $plus : \mathbb{R} \times \mathbb{Z} \to \mathbb{R}$
- Need to know arity of functions
- Instead, assume all functions take only one argument!

Functions with multiple inputs

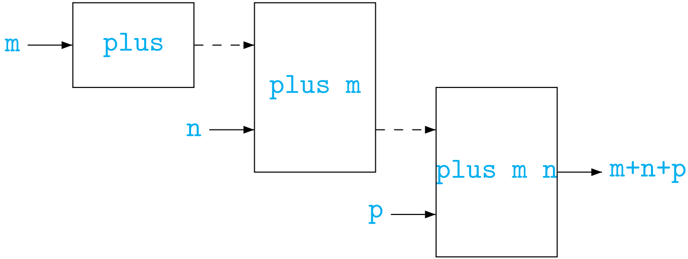
- $plus(m, n) = m + n$
  - $plus : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$, or $plus : \mathbb{R} \times \mathbb{Z} \to \mathbb{R}$
- Need to know arity of functions
- Instead, assume all functions take only one argument!



- Type of `plus`
  - `plus m`: input is `Int`, output is `Int`
  - `plus`: input is `Int`, output is a function `Int -> Int`
  - ```
    plus :: Int -> (Int -> Int)
    plus m n = m + n
    ```
    ```

Functions with multiple inputs . . .

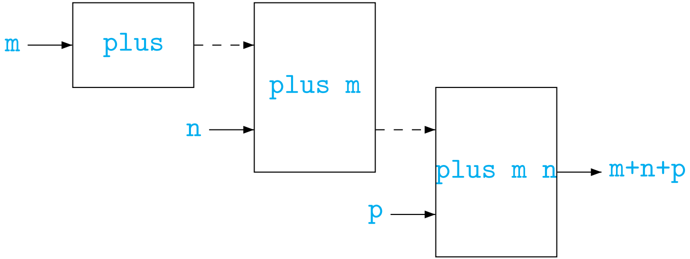
- `plus m n p = m + n + p`



- `plus m n p :: Int -> (Int -> (Int -> Int))`

Functions with multiple inputs ...

- `plus m n p = m + n + p`



- `plus m n p :: Int -> (Int -> (Int -> Int))`
- `f x1 x2 ...xn = y`
  - `x1::t1`, `x2::t2`, ..., `xn::tn`, `y::t`
  - `f::t1 -> (t2 -> ( ...(tn -> t) ...))`

Functions with multiple inputs . . .

- Function application associates to left
  - `f x1 x2 ...xn`
  - `(...((f x1) x2) ...xn)`
- Arrows in function type associate to right
  - `f :: t1 -> t2 -> ...tn -> t`
  - `f :: t1 -> (t2 -> ( ...(tn -> t) ...))`

Functions with multiple inputs ...

- Function application associates to left
  - `f x1 x2 ...xn`
  - `(...((f x1) x2) ...xn)`
- Arrows in function type associate to right
  - `f :: t1 -> t2 -> ...tn -> t`
  - `f :: t1 -> (t2 -> ( ...(tn -> t) ...))`
- Writing functions with one argument at a time = currying
  - Haskell Curry, famous logician, lends name to Haskell
  - Currying actually invented by Schönfinkel!

Defining functions

- Boolean expressions

```
xor :: Bool -> Bool -> Bool
xor b1 b2 = (b1 && (not b2)) || ((not b1) && b2)

middlebiggest :: Int -> Int -> Int -> Bool
middlebiggest x y z = (x <= y) && (z <= y)
```

Defining functions

- Boolean expressions

```
xor :: Bool -> Bool -> Bool
xor b1 b2 = (b1 && (not b2)) || ((not b1) && b2)

middlebiggest :: Int -> Int -> Int -> Bool
middlebiggest x y z = (x <= y) && (z <= y)
```

- `==`, `/=`, `<`, `<=`, `>`, `>=`, `/=`

Definition by cases: Pattern matching

- Can define `xor b1 b2` by listing out all combinations

```
if      b1 && not(b2) then True
else if not(b1) && b2 then True
else                       False
```

Definition by cases: Pattern matching

- Can define `xor b1 b2` by listing out all combinations

```
if      b1 && not(b2) then True
else if not(b1) && b2 then True
else                       False
```

- Instead, multiple definitions, with pattern matching

```
xor :: Bool -> Bool -> Bool
xor True False = True
xor False True = True
xor b1 b2 = False
```

Definition by cases: Pattern matching

- Can define `xor b1 b2` by listing out all combinations

```
if      b1 && not(b2) then True
else if not(b1) && b2 then True
else                       False
```

- Instead, multiple definitions, with pattern matching

```
xor :: Bool -> Bool -> Bool
xor True False = True
xor False True = True
xor b1 b2 = False
```

- When does an invocation match a definition?
  - If definition argument is a variable, any value supplied matches (and is substituted for that variable)
  - If definition argument is a constant, the value supplied must be the same constant

Definition by cases: Pattern matching

- Can define `xor b1 b2` by listing out all combinations

```
if      b1 && not(b2) then True
else if not(b1) && b2 then True
else                      False
```

- Instead, multiple definitions, with pattern matching

```
xor :: Bool -> Bool -> Bool
xor True False = True
xor False True = True
xor b1 b2 = False
```

- When does an invocation match a definition?
  - If definition argument is a variable, any value supplied matches (and is substituted for that variable)
  - If definition argument is a constant, the value supplied must be the same constant
- Use first definition that matches, top to bottom

Defining functions ...

- Functions are often defined inductively
  - Base case: Explicit value for $f(0)$
  - Inductive step: Define $f(n)$ in terms of $f(n-1), \ldots, f(0)$

Defining functions ...

- Functions are often defined inductively
  - Base case: Explicit value for $f(0)$
  - Inductive step: Define $f(n)$ in terms of $f(n-1), \ldots, f(0)$
- For example, factorial is usually defined inductively
  - $0! = 1$
  - $n! = n \cdot (n-1)!$

Defining functions . . .

- Functions are often defined inductively
  - Base case: Explicit value for $f(0)$
  - Inductive step: Define $f(n)$ in terms of $f(n-1), \ldots, f(0)$
- For example, factorial is usually defined inductively
  - $0! = 1$
  - $n! = n \cdot (n-1)!$
- Use pattern matching to achieve this in Haskell

```
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * (factorial (n-1))
```

Defining functions . . .

- Functions are often defined inductively
  - Base case: Explicit value for $f(0)$
  - Inductive step: Define $f(n)$ in terms of $f(n-1), \ldots, f(0)$
- For example, factorial is usually defined inductively
  - $0! = 1$
  - $n! = n \cdot (n-1)!$
- Use pattern matching to achieve this in Haskell

```
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * (factorial (n-1))
```

- Note the bracketing in `factorial (n-1)`
  - `factorial n-1` would be bracketed as `(factorial n) -1`

Defining functions . . .

- Functions are often defined inductively
  - Base case: Explicit value for $f(0)$
  - Inductive step: Define $f(n)$ in terms of $f(n-1)$,. . . , $f(0)$
- For example, factorial is usually defined inductively
  - $0! = 1$
  - $n! = n \cdot (n-1)!$
- Use pattern matching to achieve this in Haskell

```
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * (factorial (n-1))
```

- Note the bracketing in `factorial (n-1)`
  - `factorial n-1` would be bracketed as `(factorial n) -1`
- No guarantee of termination, correctness!
  - What does `factorial (-1)` generate?

Conditional definitions

- Conditional definitions using guards
- For instance, “fix” the function to work for negative inputs

```
factorial :: Int -> Int
factorial 0 = 1
factorial n
  | n < 0 = factorial (-n)
  | n > 0 = n * (factorial (n-1))
```

Conditional definitions

- Conditional definitions using guards
- For instance, "fix" the function to work for negative inputs

```
factorial :: Int -> Int
factorial 0 = 1
factorial n
  | n < 0 = factorial (-n)
  | n > 0 = n * (factorial (n-1))
```

- Second definition has two parts
  - Each part is guarded by a condition
  - Guards are tested top to bottom

Computation as rewriting

- Use definitions to simplify expressions till no further simplification is possible

Computation as rewriting

- Use definitions to simplify expressions till no further simplification is possible
- Builtin simplifications
  - `3 + 5` $\rightsquigarrow$ `8`
  - `True || False` $\rightsquigarrow$ `True`

Computation as rewriting

- Use definitions to simplify expressions till no further simplification is possible
- Builtin simplifications
  - `3 + 5` $\rightsquigarrow$ `8`
  - `True || False` $\rightsquigarrow$ `True`
- Simplifications based on user defined functions

Computation as rewriting

- Use definitions to simplify expressions till no further simplification is possible
- Builtin simplifications
  - `3 + 5` $\rightsquigarrow$ `8`
  - `True || False` $\rightsquigarrow$ `True`
- Simplifications based on user defined functions

```
power :: Float -> Int -> Float
power x 0 = 1.0
power x n | n > 0 = x * (power x (n-1))
```

Computation as rewriting

- Use definitions to simplify expressions till no further simplification is possible
- Builtin simplifications
  - `3 + 5` $\leadsto$ `8`
  - `True || False` $\leadsto$ `True`
- Simplifications based on user defined functions

```
power :: Float -> Int -> Float
power x 0 = 1.0
power x n | n > 0 = x * (power x (n-1))
```

- `power 3.0 2`

  $\leadsto$ `3.0 * (power 3.0 (2-1))`

  $\leadsto$ `3.0 * (power 3.0 1)`

  $\leadsto$ `3.0 * 3.0 * (power 3.0 (1-1))`

  $\leadsto$ `3.0 * 3.0 * (power 3.0 0)`

  $\leadsto$ `3.0 * 3.0 * 1.0`

  $\leadsto$ `9.0 * 1.0` $\leadsto$ `9.0`

Computation as rewriting . . .

- More than one expression may qualify for rewriting

Computation as rewriting . . .

- More than one expression may qualify for rewriting
- `sqr x = x*x`

Computation as rewriting . . .

- More than one expression may qualify for rewriting
- `sqr x = x*x`
- `sqr (4+3)`

  $\rightsquigarrow$ `sqr 7` $\rightsquigarrow$ `7*7` $\rightsquigarrow$ `49`

  $\rightsquigarrow$ `(4+3)*(4+3)` $\rightsquigarrow$ `(4+3)*7` $\rightsquigarrow$ `7*7` $\rightsquigarrow$ `49`

Computation as rewriting . . .

- More than one expression may qualify for rewriting
- `sqr x = x*x`
- `sqr (4+3)`

  $\leadsto$ `sqr 7` $\leadsto$ `7*7` $\leadsto$ `49`

  $\leadsto$ `(4+3)*(4+3)` $\leadsto$ `(4+3)*7` $\leadsto$ `7*7` $\leadsto$ `49`
- If there are multiple expressions to rewrite, Haskell chooses outermost expression

Computation as rewriting ...

- More than one expression may qualify for rewriting
- `sqr x = x*x`
- `sqr (4+3)`

  $\leadsto$ `sqr 7` $\leadsto$ `7*7` $\leadsto$ `49`

  $\leadsto$ `(4+3)*(4+3)` $\leadsto$ `(4+3)*7` $\leadsto$ `7*7` $\leadsto$ `49`
- If there are multiple expressions to rewrite, Haskell chooses outermost expression
- Outermost reduction $\equiv$ 'Lazy" rewriting
  Evaluate argument to a function only when needed.
- "Eager" rewriting — evaluate arguments before evaluating function

Computation as rewriting . . .

- Outermost reduction can duplicate subexpressions

Computation as rewriting . . .

- Outermost reduction can duplicate subexpressions

  `sqr (4+3)` $\rightsquigarrow$ `(4+3)*(4+3)`

Computation as rewriting ...

- Outermost reduction can duplicate subexpressions

  `sqr (4+3)` $\rightsquigarrow$ `(4+3)*(4+3)`

- Maintain pointers to identical subexpressions generated by copying at the time of reduction
  - Reduce a duplicated expression only once

Computation as rewriting ...

- Outermost reduction can duplicate subexpressions

  `sqr (4+3)` $\rightsquigarrow$ `(4+3)*(4+3)`

- Maintain pointers to identical subexpressions generated by copying at the time of reduction
  - Reduce a duplicated expression only once
- Haskell cannot otherwise detect identical subexpressions

```
diffsquare :: Float -> Float -> Float
diffsquare x y = (x - y) * (x - y)
```

Computation as rewriting . . .

- Outermost reduction may terminate when innermost does not

Computation as rewriting . . .

- Outermost reduction may terminate when innermost does not

```
power :: Float -> Int -> Float
power x 0 = 1.0
power x n | n > 0 = x * (power x (n-1))
```

- `power (7.0/0.0) 0` $\rightsquigarrow$ `1.0`

Computation as rewriting ...

- Outermost reduction may terminate when innermost does not

```
power :: Float -> Int -> Float
power x 0 = 1.0
power x n | n > 0 = x * (power x (n-1))
```

- `power (7.0/0.0) 0` $\leadsto$ `1.0`
- Outermost and innermost reduction give same answer when both terminate
  - Order of evaluation of subexpressions does not matter

Lists

- Basic collective type in Haskell is a list
  - `[1,2,3,1]` is a list of `Int`
  - `[True,False,True]` is a list of `Bool`
- Elements of a list must all be of uniform type
  - Cannot write `[1,2,True]` or `[3.0,'a']`
- List of underlying type `T` has type `[T]`
  - `[1,2,3,1]::[Int]`, `[True,False,True]::[Bool]`
- Empty list is `[]` for all types
- Lists can be nested
  - `[[3,2],[],[7,7,7]]` is of type `[[Int]]`

Internal representation on lists

- Basic list building operator is `:`
  - Append an element to the left of a list
  - `1:[2,3,4]` $\rightsquigarrow$ `[1,2,3,4]`

Internal representation on lists

- Basic list building operator is `:`
  - Append an element to the left of a list
  - `1:[2,3,4]` $\rightsquigarrow$ `[1,2,3,4]`
- All Haskell lists are built up from `[]` using operator `:`
  - `[1,2,3,4]` is actually `1:(2:(3:(4:[])))`
  - `:` is right associative, so `1:2:3:4:[]` = `1:(2:(3:(4:[])))`

Internal representation on lists

- Basic list building operator is `:`
  - Append an element to the left of a list
  - `1:[2,3,4]` $\rightsquigarrow$ `[1,2,3,4]`
- All Haskell lists are built up from `[]` using operator `:`
  - `[1,2,3,4]` is actually `1:(2:(3:(4:[])))`
  - `:` is right associative, so `1:2:3:4:[]` = `1:(2:(3:(4:[])))`
- Functions `head` and `tail` to decompose a list
  - `head (x:l) = x`
  - `tail (x:l) = l`
  - Undefined for `[]`
  - `head` returns a value, `tail` returns a list

Defining list functions inductively

- Inductive definitions
  - Define `f` for `[]`
  - Derive `f l` by combining `head l` and `f (tail l)`

Defining list functions inductively

- Inductive definitions
  - Define `f` for `[]`
  - Derive `f l` by combining `head l` and `f (tail l)`

```
length :: [Int] -> Int
length [] = 0
length l  = 1 + (length (tail l))

sum :: [Int] -> Int
sum [] = 0
sum l  = (head l) + (sum (tail l))
```

Functions on lists . . .

- Implicitly extract head and tail using pattern matching

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


sum :: [Int] -> Int
sum [] = 0
sum (x:xs) = x + (sum xs)
```

Functions on lists ...

- Combine two lists into one — `append`
  - `append [3,2] [4,6,7]` $\leadsto$ `[3,2,4,6,7]`

```
append :: [Int] -> [Int] -> [Int]
append [] ys = ys
append (x:xs) ys = x:(append xs ys)
```

Functions on lists . . .

- Combine two lists into one — `append`
  - `append [3,2] [4,6,7]` $\leadsto$ `[3,2,4,6,7]`

```
append :: [Int] -> [Int] -> [Int]
append [] ys = ys
append (x:xs) ys = x:(append xs ys)
```

- Builtin operator `++` for `append`
  - `[1,2,3] ++ [4,3]` $\leadsto$ `[1,2,3,4,3]`

Functions on lists . . .

- Combine two lists into one — `append`
  - `append [3,2] [4,6,7]` $\leadsto$ `[3,2,4,6,7]`

```
append :: [Int] -> [Int] -> [Int]
append [] ys = ys
append (x:xs) ys = x:(append xs ys)
```

- Builtin operator `++` for `append`
  - `[1,2,3] ++ [4,3]` $\leadsto$ `[1,2,3,4,3]`
- `concat` “dissolves” one level of brackets

```
concat [[Int]] -> [Int]
concat [] = []
concat (l:ls) = l ++ (concat ls)
```

- `concat [[1,2],[],[2,1]]` $\leadsto$ `[1,2,2,1]`

List functions: map

- String is a synonym for [Char]
- touppercase applies capitalize to each Char in as String
  - capitalize :: Char -> Char does what its name suggests

```
touppercase :: String -> String
touppercase "" = ""
touppercase (c:cs) = (capitalize c):(touppercase cs)
```

List functions: `map`

- `String` is a synonym for `[Char]`
- `touppercase` applies `capitalize` to each `Char` in as `String`
  - `capitalize :: Char -> Char` does what its name suggests

```
touppercase :: String -> String
touppercase "" = ""
touppercase (c:cs) = (capitalize c):(touppercase cs)
```

- An example of builtin function `map`

```
map f [x0,x1,..,xk] = [(f x0),(f x1),...,(f xk)]
```

  - Apply `f` pointwise to each element in a list

List functions: `map`

- `String` is a synonym for `[Char]`
- `touppercase` applies `capitalize` to each `Char` in as `String`
  - `capitalize :: Char -> Char` does what its name suggests

```
touppercase :: String -> String
touppercase "" = ""
touppercase (c:cs) = (capitalize c):(touppercase cs)
```

- An example of builtin function `map`

```
map f [x0,x1,..,xk] = [(f x0),(f x1),...,(f xk)]
```

  - Apply `f` pointwise to each element in a list

- `touppercase` using `map`

```
touppercase :: String -> String
touppercase s = map capitalize s
```

List functions: `map`

- `String` is a synonym for `[Char]`
- `touppercase` applies `capitalize` to each `Char` in as `String`
  - `capitalize :: Char -> Char` does what its name suggests

```
touppercase :: String -> String
touppercase "" = ""
touppercase (c:cs) = (capitalize c):(touppercase cs)
```

- An example of builtin function `map`

```
map f [x0,x1,..,xk] = [(f x0),(f x1),...,(f xk)]
```

  - Apply `f` pointwise to each element in a list

- `touppercase` using `map`

```
touppercase :: String -> String
touppercase s = map capitalize s
```

- Note that first argument of `map` is a function!
  - Higher order types

List functions: `filter`

- Select items from a list based on a property

List functions: `filter`

- Select items from a list based on a property

```
evenonly :: [Int] -> [Int]
evenonly [] = []
evenonly (n:ns)
  | mod n 2 == 0  = n:(evenonly ns)
  | otherwise     = evenonly ns
```

List functions: `filter`

- Select items from a list based on a property

```
evenonly :: [Int] -> [Int]
evenonly [] = []
evenonly (n:ns)
  | mod n 2 == 0  = n:(evenonly ns)
  | otherwise     = evenonly ns
```

- Same as applying the test

```
iseven :: Int -> Bool
iseven n = (mod n 2 == 0)
```

  to each element in the list

List functions: `filter`

- Select items from a list based on a property

```
evenonly :: [Int] -> [Int]
evenonly [] = []
evenonly (n:ns)
  | mod n 2 == 0  = n:(evenonly ns)
  | otherwise     = evenonly ns
```

- Same as applying the test

```
iseven :: Int -> Bool
iseven n = (mod n 2 == 0)
```

  to each element in the list

- `filter` selects all items from `l` that satisfy `p`

```
filter p [] = []
filter p (x:xs)
  | (p x)     = x:(filter p xs)
  | otherwise = filter p xs
```

List functions: `filter`

- Select items from a list based on a property

```
evenonly :: [Int] -> [Int]
evenonly [] = []
evenonly (n:ns)
  | mod n 2 == 0  = n:(evenonly ns)
  | otherwise     = evenonly ns
```

- Same as applying the test

```
iseven :: Int -> Bool
iseven n = (mod n 2 == 0)
```

  to each element in the list

- `filter` selects all items from `l` that satisfy `p`

```
filter p [] = []
filter p (x:xs)
  | (p x)     = x:(filter p xs)
  | otherwise = filter p xs
```

- `evenonly l = filter iseven l`

Polymorphism

- Functions like `length`, `reverse` do not need to examine elements in a list

Polymorphism

- Functions like `length`, `reverse` do not need to examine elements in a list
- Use a type variable to denote an arbitrary type

Polymorphism

- Functions like `length`, `reverse` do not need to examine elements in a list
- Use a type variable to denote an arbitrary type

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

Polymorphism

- Functions like `length`, `reverse` do not need to examine elements in a list
- Use a type variable to denote an arbitrary type

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

- Similarly
  - `reverse ::  [a] -> [a]`
  - `(++) ::  [a] -> [a] -> [a]`
  - `concat ::  [[a]] -> [a]`

Polymorphism

- Functions like `length`, `reverse` do not need to examine elements in a list
- Use a type variable to denote an arbitrary type

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

- Similarly
  - `reverse ::  [a] -> [a]`
  - `(++) ::  [a] -> [a] -> [a]`
  - `concat ::  [[a]] -> [a]`
- Polymorphism: same computation rule for multiple types

Polymorphism

- Functions like `length`, `reverse` do not need to examine elements in a list
- Use a type variable to denote an arbitrary type

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

- Similarly
  - `reverse ::  [a] -> [a]`
  - `(++) ::  [a] -> [a] -> [a]`
  - `concat ::  [[a]] -> [a]`
- Polymorphism: same computation rule for multiple types
- Overloading: same abstract operation but implementation varies
  - Representations of `Int` and `Float` are different so `+` and `*` are implemented differently

Polymorphism: `map` and `filter`

- What is the type of `map`?

  `map f [x0,x1,..,xk] = [(f x0),(f x1),...,(f xk)]`

  - Most general type for `f` is `a->b`
  - Input list is fed to `f`, so type is `[a]`
  - Output is list of items generated by `f`, so type is `[b]`

Polymorphism: `map` and `filter`

- What is the type of `map`?

  `map f [x0,x1,..,xk] = [(f x0),(f x1),...,(f xk)]`

  - Most general type for `f` is `a->b`
  - Input list is fed to `f`, so type is `[a]`
  - Output is list of items generated by `f`, so type is `[b]`
- `map :  (a -> b) -> [a] -> [b]`

Polymorphism: `map` and `filter`

- What is the type of `map`?

  `map f [x0,x1,..,xk] = [(f x0),(f x1),...,(f xk)]`

  - Most general type for `f` is `a->b`
  - Input list is fed to `f`, so type is `[a]`
  - Output is list of items generated by `f`, so type is `[b]`

- `map :  (a -> b) -> [a] -> [b]`
- What is the type of `filter`

Polymorphism: `map` and `filter`

- What is the type of `map`?

  `map f [x0,x1,..,xk] = [(f x0),(f x1),...,(f xk)]`

  - Most general type for `f` is `a->b`
  - Input list is fed to `f`, so type is `[a]`
  - Output is list of items generated by `f`, so type is `[b]`
- `map :  (a -> b) -> [a] -> [b]`
- What is the type of `filter`

```
filter p [] = []
filter p (x:xs)
  | (p x)     = x:(filter p xs)
  | otherwise = filter p xs
```

Polymorphism: `map` and `filter`

- What is the type of `map`?

  `map f [x0,x1,..,xk] = [(f x0),(f x1),...,(f xk)]`

  - Most general type for `f` is `a->b`
  - Input list is fed to `f`, so type is `[a]`
  - Output is list of items generated by `f`, so type is `[b]`
- `map : (a -> b) -> [a] -> [b]`
- What is the type of `filter`

```
filter p [] = []
filter p (x:xs)
  | (p x)      = x:(filter p xs)
  | otherwise = filter p xs
```

- `filter : (a -> Bool) -> [a] -> [a]`

Summary

- Haskell: a notation for defining computable functions
  - Currying to deal with functions of different arities
- Computation is rewriting
  - Haskell uses outermost reduction
  - Order of evaluation does not change the answer (if an answer is produced!)
- Higher order types
  - Can pass functions as arguments
- Polymorphism
  - Same rule works for multiple types