MATHFUN Lecture FP6
Functions as Values

Matthew Poole

`moodle.port.ac.uk`

School of Computing
University of Portsmouth

2014/15

Introduction to Lecture

- In this lecture we see some powerful features of functional programming rarely seen in imperative programming languages.
- We see that **functions can be treated as data**, so:
    - they can be passed as arguments to other functions; and
    - they can be returned as results from functions.
- A function that either takes another function as an argument, or returns a function, is known as a **higher-order function**.
- As we'll see, higher-order programming can be very expressive.
- E.g., we'll see how many tasks which require primitive recursion can be written very concisely (i.e. using very little code).
- This lecture is supported by Chapters 10 & 11 of Thompson's book.

Functions as arguments

- Let's begin with a simple example function definition where a function is used as an argument.
- The following function definition applies a function to a value, and then applies the same function again to the result:

```
twice :: (Int -> Int) -> Int -> Int
twice f x = f (f x)
```

- Note that type of the first argument is `Int -> Int` (i.e. function from `Int` to `Int`).

Functions as arguments

- Suppose that we have defined a function:

```
succ :: Int -> Int
succ n = n + 1
```

- An example calculation of an expression involving `twice` is:

```
twice succ 5
⇝ succ (succ 5)        def of twice
⇝ succ (5 + 1)         def of succ
⇝ succ 6               arithmetic
⇝ 6 + 1                def of succ
⇝ 7                    arithmetic
```

Function composition

- Haskell includes a function composition operator "."
- For two functions `f` and `g`, the expression:

  ```
  (f . g) x
  ```

  means apply `g` to `x`, and then apply `f` to the result.
- In other words, the effect of `(f . g)` is given by:

  ```
  (f . g) x = f (g x)
  ```

- The type of `.` is given by:

  ```
  ghci> :type (.)
  (.) :: (b -> c) -> (a -> b) -> a -> c
  ```

- Notice that the output type of the first function to be applied must be the same as the input type of the second function.

Function-level definitions

- The composition operator allows us to easily define functions just in terms of other functions.
- For example, we can replace the definition:

  ```
  twice f x = f (f x)
  ```

  by:

  ```
  twice f = f . f
  ```

- A function defined just in terms of other functions (without reference to other args) is known as a **function-level definition**.
- Another, trivial, function-level definition is:

  ```
  multiply = (*)
  ```

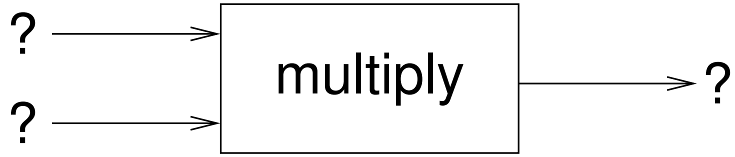
  which provides an alternative name (as a function) for the multiplication operator.

Partial application

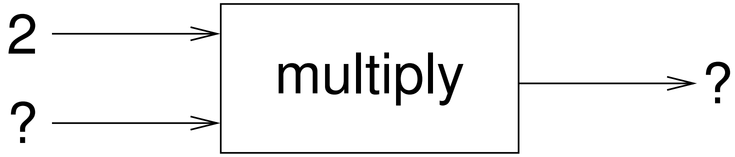
- One of the most powerful features of Haskell is that functions can be **partially** applied.
- Consider the following simple function:

```
multiply :: Int -> Int -> Int
multiply x y = x * y
```

- The standard view of this function is that it takes two arguments, and gives a result value:



- However, a more correct view is that when we apply it to one argument we are left with a function of one argument:



Partial application

- We might use the partial application of `multiply` as follows:

  ```
  double = multiply 2
  ```

  so that:

  ```
  ghci> double 5
  10
  ```

- (Question: what's the type of double?)
- Every Haskell function in fact takes exactly one argument, and the type:

  ```
  multiply :: Int -> Int -> Int
  ```

  is actually shorthand for:

  ```
  multiply :: Int -> (Int -> Int)
  ```

- (i.e. the `->` symbol is right associative.)

Operator sections

- Operators can also be partially applied; the following are examples of **operator sections**:
  - `(2*)` – a function that multiplies its argument by 2 (equivalent to `multiply 2`).
  - `(/2)` – a function that divides its argument by 2.
  - `(2/)` – a function which gives 2 divided by its argument.
  - `(>3)` – a function which tests if its argument is greater than 3.
- Questions ... what are the following functions?
  - `(3>)`
  - `(++"\n")`
  - `(7:)`
- As we'll now see, operator sections are often used together with some standard higher-order functions from the Prelude.

Patterns of computation

- Last week we looked at (primitive) recursive functions that operated over lists, and gave several examples of such functions.
- In writing list-processing functions, several computational **patterns** are common; we often want to:
  - transform every element of a list in some way;
  - remove those elements of a list that don't possess a given property; and
  - combine all the elements with a particular operation.
- We'll look at three higher-order functions that implement each of these patterns (and therefore simplify list processing).

Applying to all — mapping

- Many functions transform elements of a list in some way. E.g.:
  - Doubling all the elements of a list:

```
doubleAll :: [Int] -> [Int]
doubleAll []       = []
doubleAll (x:xs)   = 2*x : doubleAll xs
```

  - Testing whether the elements are digits:

```
areDigits :: [Char] -> [Bool]
areDigits []       = []
areDigits (x:xs)   = isDigit x : areDigits xs
```

- These functions share a common pattern: the difference (shown underlined) is the operation that is applied to the elements.
- (Note that both of these functions could be defined using list comprehensions, again with similar structures.)

Applying to all – mapping

- The Prelude defines a function `map` which takes the operation to be applied as a parameter.
- We give a definition of `map` as:

```
map :: (a -> b) -> [a] -> [b]
map f []       = []
map f (x:xs)   = f x : map f xs
```

- Using map, the definition of the functions `doubleAll` and `areDigits` become (note the use of an operator section):

```
doubleAll xs = map (*2) xs
areDigits xs = map isDigit xs
```

- Questions:
  - These definitions can be further simplified – how?
  - Using `map`, write a function that capitalises a string.

Filtering

- A second common operation on lists is **filtering**: keeping only those elements that have a certain property. Examples:
  - To keep only positive numbers:

```
keepPositive []        =  []
keepPositive (x:xs)
     | x > 0           = x : keepPositive xs
     | otherwise       = keepPositive xs
```

  - To keep only digits:

```
keepDigits []        =  []
keepDigits (x:xs)
     | isDigit x       = x : keepDigits xs
     | otherwise       = keepDigits xs
```

- Again, these functions share a common pattern. (They could also be more easily expressed as list comprehensions.)

Filtering

- The Prelude defines a function `filter` which takes the "property" as a parameter; we may define `filter` as:

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

- The functions `keepPositive` now become:

```
keepPositive = filter (>0)
keepDigits = filter isDigit
```

- (Note that here we have given function-level definitions.)
- Q: Define a fn which removes odd numbers from a list of ints.

Combining elements – folding

- A final pattern is **combining** the elements of a list in some way (known as **folding**). Examples:
  - Summing the elements of a list:

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

  - Concatenating a list of lists into one list:

```
concat :: [[a]] -> [a]
concat []       = []
concat (x:xs)   = x ++ concat xs
```

- There are two differences between these functions:
  - the function to apply (a binary function—it has 2 args); &
  - the value to return for an empty list.

Combining elements – folding

- The Prelude includes a fn `foldr` which takes the function to be applied, and the value to be returned for the empty list.
- We can define `foldr` as:

```
foldr :: (a -> a -> a) -> a -> [a] -> a
foldr f s []        = s
foldr f s (x:xs)    = f x (foldr f s xs)
```

- Using `foldr`, we can re-define `addUp` and `concat` as follows:

```
addUp = foldr (+) 0
concat = foldr (++) []
```

- Questions:
  - What does the function: `mystery = foldr (&&) True` do?
  - Define a function `multUp` which finds the product of the elements of a list.