Modern Programing Languages

Introduction to Haskell (2)

Wamberto Vasconcelos
w.w.vasconcelos@abdn.ac.uk

2016–2017

Plan of Lecture

1. Lists
2. Polymorphic Types
3. Higher-Order Functions
4. Folding a List of Values
5. Curried Functions
6. Partial Application and Operator Sections
7. Computing with Infinite Data Structures

Lists

- A list is an ordered collection of values.
- All elements must be of the same type!
- Examples:

```
           [2,6,3,7]  ::  [Int]
   ['2','6','3','7']  ::  [Char]
              "2637"  ::  [Char]
[(1,'a'),(2,'b'),(3,'c')]  ::  [(Int,Char)]
```

- N.B.: `[3,7,True,4]` is not a valid list: it contains values of different types.
- A list with no elements is called the empty list, written as "`[]`".

Lists (2)

- Haskell provides shorthands for lists of numbers:
  - `[n..m]` stands for $[n, n+1, \ldots, m']$; if $n > m$, then the result is `[]`.
  - `[n,p..m]` stands for $[n, n+k, n+k+k, \ldots, m']$, where $k = p - n$.
- m': the largest/smallest number in the sequence which is greater/less than or equal to `m` (i.e. it does not go after `m`)
- Examples:

```
       [10..15]  ⇝  [10,11,12,13,14,15]
      [10,8..0]  ⇝  [10,8,6,4,2,0]
[0,0.6  ..   2]  ⇝  [0,0.6,1.2,1.8]
```

Some Built-In Operations on Lists

- Adding an element to a list (cons) "`:`":

  `(1:(2:(3:[])))` $\leadsto$ `[1,2,3]`

- Concatenating two lists "`++`":

  `[1,2,3] ++ [4,5,6]` $\leadsto$ `[1,2,3,4,5,6]`

- Finding the length of a list "`length`":

  `length []` $\leadsto$ `0`
  `length ['a','b','c']` $\leadsto$ `3`

- Obtaining the first element (head) of a list and the remaining elements (tail):

  `head [1,2,3,4]` $\leadsto$ `1`
  `tail [1,2,3,4]` $\leadsto$ `[2,3,4]`

Pattern Matching on Lists

- The pattern "`(x:xs)`" splits a list into its first element, "`x`", and everything else, "`xs`".

  ```
  last [x] = x
  last (x:xs) = last xs
  ```

- In recursive functions over lists
  - base case is "`[x]`" or "`[]`";
  - recursive case relates the solution for "`x`" and the solution for "`xs`"
- Example:

  ```
  zip [] [] = []
  zip (x:xs) (y:ys) = (x,y):(zip xs ys)
  ```

Polymorphic Types in Haskell (1)

- What is the type of `zip`?
- Some functions have general definitions for lists of any type!
- To express this, we must use a type variable.
- In Haskell, type variables are specified using conventioanl variables as "`a`", "`my_type`", etc.
- Example:

```
zip ::  [a] -> [b] -> [(a,b)]
```

- Types specified as type variables are called polymorphic types (many forms).

Polymorphic Types in Haskell (2)

- When applying a function with polymorphic types to particular arguments, the specific types of the arguments give values to the type variables:

  `zip [1,2] ['a','b']` $\rightsquigarrow$ `[(1,'a'),(2,'b')]`

- Variable `a` is instantiated to type `Int` and `b` to `Char`, that is:

  `[Int] -> [Char] -> [(Int,Char)]`

- Types without type variables are called monomorphic types.
- Traditional languages, such as C, only support monomorphic types (though Java now has "generic types")

Type Synonyms in Haskell

- To improve the readability of programs, Haskell allows us to give names to (complex) types:

```
type String = [Char]
type Date = (Int,String,Int)
tomorrow ::  Date -> Date
```

- We can also parameterise type synonyms using type variables:

```
type ThreeTuple a = (a,a,a)
primaryColours ::  ThreeTuple String
primaryColours = ("red","green","blue")
```

Higher-Order Functions (1)

- A function is a mapping from elements of one type to elements of another type.
- Haskell places no constraints on what those types may be.
- In particular, Haskell allows us to define functions which take other functions as arguments and/or return functions as their result
- Such functions are said to be higher order. Example:

```
double ::  (a -> a) -> a -> a
double f x = f (f x)
```

- This function applies the function `f` (an argument!) to `x` twice, that is,

$$\texttt{double square 2} \rightsquigarrow \texttt{16}$$

Higher-Order Functions (2)

- We can generalise the previous functon and define function composition ".":

```
(.)   ::   (a -> b) -> (b -> c) -> a -> c
(f.g) x = f (g x)
```

- Function composition is useful for stringing function applications together:

```
greatGrannies = mothers.parents.parents
```

- We can now define `double` more easily as:

```
double f = (f.f)
```

Programming with Higher-Order Functions (1)

- Higher-order functions capture common patterns of computation (idioms).

```
squares []      = []
squares (x:xs)  = (square x):(squares xs)

allUpper []     = []
allUpper (x:xs) = (upper x):(allUpper xs)
```

- All functions exhibit the typical recursive idiom for transforming elements of a list.
- Idioms can be abstracted as higher order function definitions:

```
map f [] = []
map f (x:xs) = (f x):(map f xs)
```

- What is the type of `map`?

Programming with Higher-Order Functions (2)

- We can rewrite the previous functions in terms of `map`:

```
squares xs   = map square xs
allUpper xs  = map upper xs
```

- The type of `map` is:

```
map ::  (a -> b) -> [a] -> [b]
```

Selecting Items from Lists (1)

- Common idiom in list processing: selection of elements.

```
onlyUpper [] = []
onlyUpper (x:xs)
   | isUpper x   = x:(onlyUpper xs)
   | otherwise   = onlyUpper xs

bignums n [] = []
bignums n (x:xs)
   | big n x     = x:(bignums n xs)
   | otherwise   = bignums n xs
```

Selecting Items from Lists (2)

- This idiom is abstracted by the higher-order function `filter`:

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

- We can use `filter` to define the previous functions as:

```
onlyUpper xs = filter isUpper xs
bignums n xs = filter (big n) xs
```

- The type of `filter` is:

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

Folding a List of Values (1)

- A more complex but very versatile idiom consists of folding a binary function into a list of values to compute a single value.
- Example: to add a list of numbers, fold in the `+` operator, that is,

  `sum [5,2,8,14,11]` $\rightsquigarrow$ `5+2+8+14+11`
- Example: to find the conjunction of a list of Boolean values, fold in operator `&`, that is,

  `and [True,False,True]` $\rightsquigarrow$ `True & False & True`
- Haskell definitions for these functions:

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

and [] = True
and (x:xs) = x & (and xs)
```

Folding a List of Values (2)

- The abstraction of this idiom is defined as:

```
foldr ::  (a -> b -> b) -> b -> [a] -> b

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

- `foldr` defines `sum` and `and` as follows:

```
sum xs = foldr (+) 0 xs

and xs = foldr (&) True xs
```

- This function is called `foldr` because it brackets the folded expression to the right, that is,

  `5+(2+(8+(14+11)))`

  and not `(((5+2)+8)+14)+11`

Folding a List of Values (3)

- We can also define a second version of the `fold` idiom, called `foldl`, which brackets the folded expression to the left:

  ```
  foldl ::  (a -> b -> a) -> a -> [b] -> a
  ```

  ```
  foldl f a [] = a
  foldl f a (x:xs) = foldl f (f a x) xs
  ```

- The types of `foldr` and `foldl` are subtly different.
- This is the main factor which influences the choice of which of these functions to use in a given situation.
- If the function you wish to fold into the list has a type which matches `(a -> b -> b)` then you should use `foldr`. An example is:

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

Folding a List of Values (4)

- If the function's type matches `(a -> b -> a)`, then you should use `foldl`.
- The constant function has exactly this type: `const x y = x`
- When applied to a function `f` with a type matching `(a -> a -> a)`, `foldr` and `foldl` have the same type, that is,

  `(a -> a -> a) -> a -> [a] -> a`
- If `f` is also associative, then `foldr f a xs` $\equiv$ `foldl f a xs`
- In this case, the choice must be made on the basis of which of these functions gives the most efficient solution, which will depend upon the characteristics of `f`.

Folding a List of Values (5)

- The folding idiom can be used to define a surprisingly diverse range of functions.
- Example – reversing a list:

```
reverse ::  [a] -> [a]
reverse xs = foldr rev [] xs
    where rev a bs = bs ++ [a]
```

- Example – finding the maximum element of a list:

```
maximum ::  [Int] -> Int
maximum (x:xs) = foldl max x xs
    where max n m
           | n >= m    = n
           | otherwise = m
```

Curried Functions in Haskell (1)

- Consider the function:

```
plus ::  Int -> Int -> Int
plus x y = x + y
```

- The "`->`" operator for declaring function types associates to the right.
- So, strictly speaking the type of the function should be:

```
plus ::  Int -> (Int -> Int)
```

- But this says that `plus` is a function which takes a number as its (only) argument and returns a function of type `(Int -> Int)` as its result!!
- What is the meaning of, for example, `plus 2`?

```
(plus 2) y = 2 + y
```

Curried Functions in Haskell (2)

- So, strictly speaking, `plus` is a function which takes a number, `x`, as its only argument and constructs as its result a function which takes another number `y` as an argument and adds `x` to it.
- This technique is known as Currying (after the logician Haskell B. Curry).
- Currying allows us to define functions with multiple arguments without departing from the true mathematical definition of a function as a mapping from one set to another.
- Currying results in more readable definitions than if we continually had to bundle multiple arguments up into tuples.
- Currying also makes higher-order programming much easier, as we do not need to know how many arguments a function requires in order to apply it.

Partial Application and Operator Sections (1)

- The application of a function requiring n arguments to fewer than n values is called partial application.
- Partial application is a useful way of building new nameless functions.
- The same technique can be applied to operators:

  `(+2)` `(=0)` `(*3)` `(>20)`

- The result is called an operator section.
- Functions like these are particularly handy as arguments to `map` and `filter`, though *not all work with current Hugs*.
- Example:

```
filter (>5) [2,4,6,8,10] ⇝ [6,8,10]
map (*2) [1,2,3] ⇝ [2,4,6]
```

Partial Application and Operator Sections (2)

- The brackets are important here.
- They turn an infix operator into a prefix operator:

$$\texttt{x + y} \equiv \texttt{(+) x y}$$

- In the same way, a prefix function can be turned into an infix operator if we enclose it in backquotes ‘...‘, that is,

$$\texttt{map f l} \equiv \texttt{f `map` l}$$

Computing with Infinite Data Structures (1)

- One of the advantages of a lazy evaluation strategy is that we can compute with infinite data structures.
- For example:

```
ones ::  [Int]
ones = 1 :  ones
```

- What is the result of evaluating `head ones`?
- An eager evaluation strategy will not terminate when presented with this expression.
- A lazy evaluation strategy will compute only as much of the infinite list of ones as is needed to evaluate `head`.
- Its motto is "never do today what you can put off until asked tomorrow".

Computing with Infinite Data Structures (1)

- Example 1:

```
-- All natural numbers
nn = [1..]
```

- Example 2:

```
-- The squares of all natural numbers
nnSquares = map square nn
   where square x = x * x
```

- Example 3:

```
-- The Fibonacci series
all_fib = 1:(1:(add_fib all_fib))
   where add_fib(x:(y:rs)) = (x+y):(add_fib (y:rs))
```

Computing with Infinite Data Structures (2)

- What happens when Haskell tries to evaluate `all_fib`?
- Note that we start by cons-ing together the first two members of the series.
- We now have just enough to calculate the third member (`x+y = 1+1 = 2`), should anybody ask for it.

Computing with Infinite Data Structures (3)

- To manipulate infinite lists, we make use of functions which produce a finite result from an infinite list:

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

- Examples:

```
take 10 [1..] ⇝ [1,2,3,4,5,6,7,8,9,10]
take 5 nnSquares ⇝ [1, 4, 9, 16, 25]
nnSquares !!  4 ⇝ 25
filter (<20) nnSquares ⇝ [1, 4, 9, 16]
filter (<20) (filter (>10) nnSquares) ⇝ [16]
```