Outline of Lecture 7

- Patterns of computation
- Higher order functions
- Function composition and application

Generalisation: Patterns of computation

One mechanism to generalise computations in Haskell – to define polymorphic functions that work for all or many different concrete types

Another one is to implement reusable patterns of computations.
For instance, examples of such patterns for working with lists would be:

- Transform every element of a list in some way;
- Select/ filter/ break up a list into smaller parts using some criteria;
- Combine the elements of a list using some operator;
- A mixture of the above.

This can be achieved via creating higher-order functions, i.e., functions that take other functions (the particular kind of computations) as parameters

Higher-order functions: Mapping

Mapping (transforming every element of a list in some way)

- The `map` function:

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

- The first parameter – a transformation function (from arbitrary type `a` to a possibly different type `b`)
- The second parameter – the list to be transformed
- General definition of `map`:

```
map f xs = [f x | x <- xs]
```

- Convenient to have such a definition in the explicit function form to possibly combine it with other functions

Mapping examples

- Doubling list elements:

```
doubleAll ::  [Integer] -> [Integer]
doubleAll xs = map double xs

double ::  Integer -> Integer
double x = 2*x
```

- Converting characters into their codes:

```
convertChrs ::  [Char] -> [Int]
convertChrs xs = map fromEnum xs
```

Mapping examples (combining zip and map)

- Polymorphic function zip:

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

- What if we want to do something different (other to just pairing them up) to two corresponding list elements:

```
zipWith ::  (a->b->c) -> [a] -> [b] -> [c]
zipWith f (x:xs) (y:ys) = f x y :  zipWith f xs ys
zipWith f _ _ = []
```

- A version relying on map:

```
zipWith' ::  ((a,b)->c) -> [a] -> [b] -> [c]
zipWith' f xs ys = map f (zip xs ys)
```

Higher-order functions: Filtering

Filtering a list according to the given property

- How a property (e.g., of being a digit, an even number, etc.) is expressed in Haskell?
- In general, a property is given as a function of the type `t -> Bool`, where `t` is a concrete type we are dealing with, e.g., `isDigit :: Char -> Bool`, `isEven :: Integer -> Bool`
- The `filter` function:

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

  where the first argument is a property function and the second argument is the list to be filtered

Higher-order functions: Filtering (cont.)

- General definition of `filter`:

```
filter p xs = [x | x <- xs, p x]
```

- Examples:

```
digits ::  String -> String
digits xs = filter isDigit xs

evenNumbers ::  [Integer] -> [Integer]
evenNumbers xs = filter isEven xs

sortedLists ::  [[Integer]] -> [[Integer]]
sortedLists xs = filter isSorted xs

isSorted ::  [Integer] -> Bool
isSorted xs = (xs == iSort xs)
```

Variations of filtering

- Skipping or continuing until/while some property holds

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

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

- `takeWhile (dropWhile)` – predefined in Haskell

Higher-order functions: Folding

- Combining (*folding*) elements according to the given function
- The folding function is applied for each list element, combining its value with the previously accumulated result. The calculation is performed either from the list beginning or the list end
- Many primitive recursive functions over lists can be defined via folding
- Several varieties of higher-order functions based on folding
- Folding (to the right) of a non-empty list:

```
foldr1 ::  (a->a->a) -> [a] -> a
foldr1 f [x] = x
foldr1 f (x:xs) = f x (foldr1 f xs)
```

Higher-order functions: Folding (cont.)

- The first argument: a binary function over an arbitrary type `a`
- The second argument: the list to be combined (folded) together into one value
- What does it mean "folding to the right"?

```
   foldr1 f [e1,e2, e3, ..., en]
        = e1 'f' (e2 'f' ( e3 'f' (...  'f' en) ...)))
```

  From the list beginning to the list end

Higher-order functions: Folding (cont.)

Folding examples (with `foldr1`):

```
foldr1 (+) [3,98,1] = 102
foldr1 (||) [False,True,False] = True
foldr1 min [6] = 6
foldr1 (*) [1 ..  6] = 720
```

Drawback: `foldr1` is not defined for `[]`

Higher-order functions: Folding (cont.)

- (Slightly) generalised folding to the right:

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

- The first argument: a binary function over an arbitrary type a
- The second argument: the starting (default) value of the type a
- The third argument: the list to be folded
- The definition of `foldr`:

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

Higher-order functions: Folding (cont.)

Using `foldr`, we can define now some standard functions of Haskell, like concatenating a list of lists or calculating conjunction over a list of boolean values:

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

and ::  [Bool] -> Bool
and bs = foldr (&&) True bs
```

`foldr1` can be then defined in terms of `foldr`:

```
foldr1 f xs = foldr f (last xs) (init xs)
```

where `last` gives the last element of a list, and `init` removes that element

Primitive recursion and folding

Often, we can replace primitive recursion over a list with folding. For example, sorting by insertion is defined (see the Lecture 5 slides)

```
iSort [] = []
iSort (x:xs) = ins x (iSort xs)
```

We could rewrite the second equation as

```
iSort (x:xs) = x `ins` (iSort xs)
```

and the whole definition as

```
iSort xs = foldr ins [] xs
```

The problem: the type of `ins` is `::a -> [a] -> [a]`, not `::a -> a -> a`, as required by `foldr`

Higher-order functions: Folding (cont.)

- No problem, since the actual type of `foldr` is even more general
- Most general folding to the right:

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

- The resulting folding type can be different! That means that the above definition of `iSort` via `foldr` is correct (the type `a` stays `a`, while the type `b` becomes `[a]`)

Higher-order functions: Folding (cont.)

Another example: redefining list reversion via folding

```
rev ::  [a] -> [a]
rev xs = foldr snoc [] xs

snoc ::  a -> [a] -> [a]
snoc x xs = xs ++ [x]
```

The function `snoc` creates a list by always adding an element to the list end, i.e., does the opposite of `':'` operator

The functions `foldr1` and `foldr` have their dual versions `foldl1` and `foldl` that define folding to the left, i.e., from the list end towards its beginning

Functional composition

- Functional composition is one of the simplest ways of structuring a functional program by composing together separate functions
- Functional composition has an effect of wiring two functions together by passing the output of one to the input of the other. As a result, a new composite function is automatically created
- It is defined as the operator `(.)`

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

- First execute `g::a->b`, then pass its output to `f::b->c`, and execute `f`. The resulting function `(f . g)` is of the type `::a->c`

Function composition: examples

Applying to arguments:

```
> (negate . abs) 5
-5
> (negate . abs) (-7)
-7
> map (negate . sum . tail) [[1..5],[3..6],[1..7]]
[-14,-15,-27]
```

Defining new functions:

```
makeNegatives ::  [Integer] -> [Integer]
makeNegatives = map (negate . abs)

fun ::  Integer -> Integer
fun = sum . (replicate 8) . negate
```

Function application

- The typical use of function application: `f x`
- In Haskell, there is also the explicit operator of function application:

```
($) ::  (a->b) -> a -> b
f $ e = f e
```

- Why is it needed?
  - Many Haskell programmers use $ as alternative to parentheses, e.g., instead of writing

    ```
    flipV (flipH (rotate horseFig))
    ```

    it is possible to write

    ```
    flipV $ flipH $ rotate horseFig
    ```

  - Sometimes we need the application operator as a function that, e.g. we pass as an argument

    ```
    zipWith ($) [sum, product] [[1,2],[3,4]]
    ```