FP101x - Functional Programming

Programming in Haskell – Recursive Functions

Erik Meijer

Introduction

As we have seen, many functions can naturally be defined in terms of other functions.

```
factorial  :: Int → Int
factorial n = product [1..n]
```

factorial maps any integer n to the product of the integers between 1 and n.

Expressions are <u>evaluated</u> by a stepwise process of applying functions to their arguments.

For example:

factorial 4

=

product [1..4]

=

product [1,2,3,4]

=

1*2*3*4

=

24

Recursive Functions

In Haskell, functions can also be defined in terms of themselves. Such functions are called <u>recursive</u>.

```
factorial 0 = 1
factorial n = n * factorial (n-1)
```

factorial maps 0 to 1, and any other integer to the product of itself and the factorial of its predecessor.

For example:

```
  factorial 3
= 3 * factorial 2
= 3 * (2 * factorial 1)
= 3 * (2 * (1 * factorial 0))
= 3 * (2 * (1 * 1))
= 3 * (2 * 1)
= 3 * 2
= 6
```

Note:

- factorial 0 = 1 is appropriate because 1 is the identity for multiplication: 1*x = x = x*1.

- The recursive definition <u>diverges</u> on integers < 0 because the base case is never reached:

```
> factorial (-1)

Exception: stack overflow
```

Why is Recursion Useful?

Some functions, such as factorial, are <u>simpler</u> to define in terms of other functions.

As we shall see, however, many functions can <u>naturally</u> be defined in terms of themselves.

Properties of functions defined using recursion can be proved using the simple but powerful mathematical technique of <u>induction</u>.

Recursion on Lists

Recursion is not restricted to numbers, but can also be used to define functions on lists.

```
product         :: [Int] → Int
product []      = 1
product (n:ns) = n * product ns
```

product maps the empty list to 1, and any non-empty list to its head multiplied by the product of its tail.

For example:

```
product [2,3,4]
```
=
```
2 * product [3,4]
```
=
```
2 * (3 * product [4])
```
=
```
2 * (3 * (4 * product []))
```
=
```
2 * (3 * (4 * 1))
```
=
```
24
```

Using the same pattern of recursion as in product we can define the <u>length</u> function on lists.

```
length         :: [a] → Int
length []       = 0
length (_:xs) = 1 + length xs
```

length maps the empty list to 0, and any non-empty list to the successor of the length of its tail.

For example:

```
length [1,2,3]
```
=
```
1 + length [2,3]
```
=
```
1 + (1 + length [3])
```
=
```
1 + (1 + (1 + length []))
```
=
```
1 + (1 + (1 + 0))
```
=
```
3
```

Using a similar pattern of recursion we can define the reverse function on lists.

```
reverse         :: [a] → [a]
reverse []       = []
reverse (x:xs) = reverse xs ++ [x]
```

reverse maps the empty list to the empty list, and any non-empty list to the reverse of its tail appended to its head.

For example:

```
reverse [1,2,3]
```
=
```
reverse [2,3] ++ [1]
```
=
```
(reverse [3] ++ [2]) ++ [1]
```
=
```
((reverse [] ++ [3]) ++ [2]) ++ [1]
```
=
```
(([] ++ [3]) ++ [2]) ++ [1]
```
=
```
[3,2,1]
```

Multiple Arguments

Functions with more than one argument can also be defined using recursion. For example:

- Zipping the elements of two lists:

```
zip              :: [a] → [b] → [(a,b)]
zip []      _        = []
zip _       []       = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

Remove the first n elements from a list:

```
drop            :: Int → [a] → [a]
drop 0 xs       = xs
drop _ []       = []
drop n (_:xs) = drop (n-1) xs
```

Appending two lists:

```
(++)            :: [a] → [a] → [a]
[]       ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

Quicksort

The quicksort algorithm for sorting a list of integers can be specified by the following two rules:

- The empty list is already sorted;

- Non-empty lists can be sorted by sorting the tail values ≤ the head, sorting the tail values > the head, and then appending the resulting lists on either side of the head value.
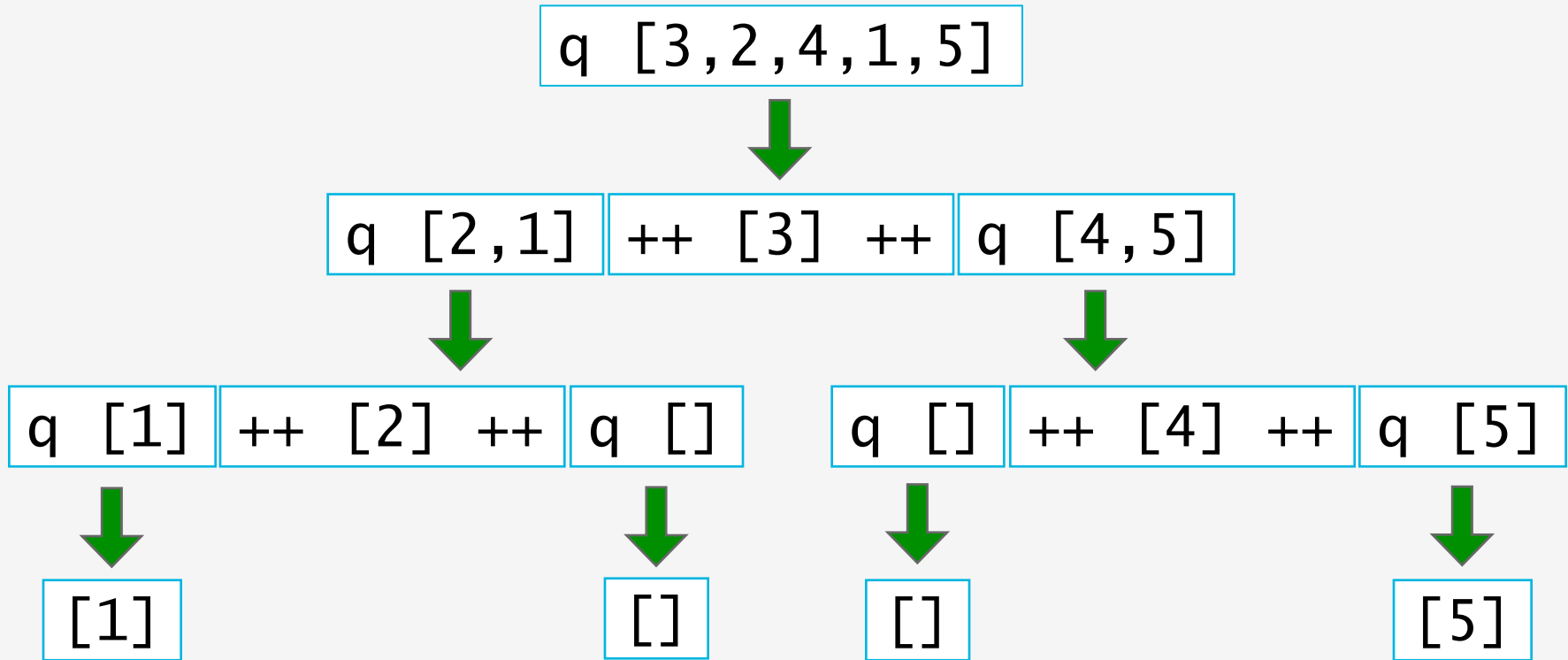
Using recursion, this specification can be translated directly into an implementation:

```
qsort        :: [Int] → [Int]
qsort []      = []
qsort (x:xs) =
    qsort smaller ++ [x] ++ qsort larger
    where
        smaller = [a | a ← xs, a ≤ x]
        larger  = [b | b ← xs, b > x]
```

Note:

- This is probably the <u>simplest</u> implementation of quicksort in any programming language!

For example (abbreviating qsort as q):

```
q [3,2,4,1,5]
```

```
q [2,1]   ++ [3] ++   q [4,5]
```

```
q [1]   ++ [2] ++   q []        q []   ++ [4] ++   q [5]
```

```
[1]              []        []              [5]
```

Exercises

(1) Without looking at the standard prelude, define the following library functions using recursion:

■ Decide if all logical values in a list are true:

```
and :: [Bool] → Bool
```

■ Concatenate a list of lists:

```
concat :: [[a]] → [a]
```

Produce a list with n identical elements:

```
replicate :: Int → a → [a]
```

Select the nth element of a list:

```
(!!) :: [a] → Int → a
```

Decide if a value is an element of a list:

```
elem :: Eq a ⇒ a → [a] → Bool
```

(2)   Define a recursive function

```
merge :: [Int] → [Int] → [Int]
```

that merges two sorted lists of integers to give a single sorted list. For example:

```
> merge [2,5,6] [1,3,4]

[1,2,3,4,5,6]
```

(3) Define a recursive function

```
msort :: [Int] → [Int]
```

that implements <u>merge sort</u>, which can be specified by the following two rules:

■ Lists of length ≤ 1 are already sorted;

■ Other lists can be sorted by sorting the two halves and merging the resulting lists.

Happy Hacking!