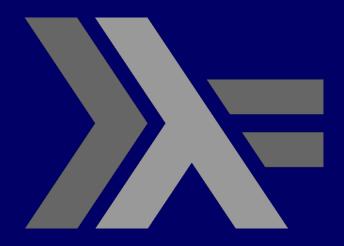
PROGRAMMING IN HASKELL



Chapter 6 - Recursive Functions

Introduction

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

```
fac :: Int \rightarrow Int
fac n = product [1..n]
```

fac 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:

```
fac 4
product [1..4]
product [1,2,3,4]
1*2*3*4
```

Recursive Functions

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

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

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

For example:

```
fac 3
   fac 2
       * fac 1)
             * fac 0))
          (1
       *
       *
6
```

Note:

□ fac 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:</p>

```
> fac (-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 naturally 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 <u>lists</u>.

```
product :: Num a \Rightarrow [a] \rightarrow a
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]
  * product [3,4]
    (3 * product [4])
       * (4 * product []))
       * (4 * 1))
```

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

```
length :: [a] \rightarrow 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))
```

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]
```

Looking at insertion sort, we first look at a function that inserts a new element of any Ordered type into a sorted list to give another sorted list.

```
insert :: Ord a \Rightarrow a \rightarrow [a] \rightarrow [a]
insert x [] = [x]
insert x (y:ys)
| x \le y = x : y : ys
| otherwise = y: insert x ys
```

Now, using insert, we can define a function that implements

Insertion sort

in which the empty list is already sorted and any non-empty list is sorted by inserting its head into the list that results from sorting its tail.

```
isort :: Ord a \Rightarrow [a] \rightarrow [a]
isort [] = [x]
isort (x:xs) = insert x (isort xs)
```

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 _ (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] \rightarrow [a] \rightarrow [a]
[] ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

Quicksort

The <u>quicksort</u> algorithm for sorting a list of values 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.

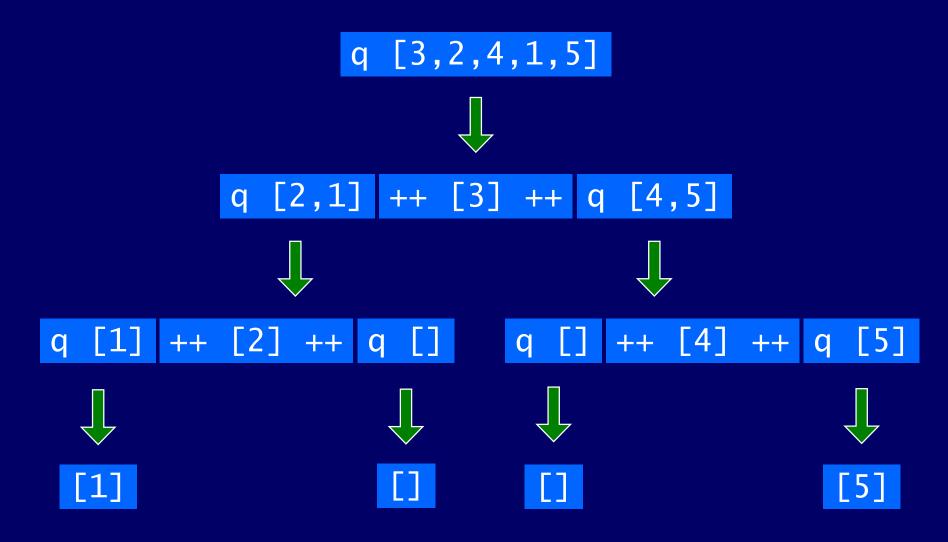
Using recursion, this specification can be translated directly into an implementation:

```
qsort :: Ord a \Rightarrow [a] \rightarrow [a]
qsort [] = []
qsort (x:xs) =
   qsort smaller ++ [x] ++ qsort larger
   where
   smaller = [a | a \leftarrow xs, a \leq x]
   larger = [b | b \leftarrow xs, b > x]
```

Note:

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

For example (abbreviating qsort as q):



Advice on Recursion

- □ Step 1 Define the type,
- ☐ Step 2 Enumerate the cases,
- □ Step 3 Define the simple case,
- □ Step 4 Define the other cases,
- Step 5 Generalise and simplify.

Example 1 – product using steps

product – the function that takes a list and returns the product of the numbers.

Step 1 – Define the type

product :: [Int] -> Int

(Use the simplest type possible)

Step 2 - Enumerate the cases

For most types, there are a number of standard cases to consider.

For lists, these are

- the empty list and
- ☐ the non-empty list

```
product [] =
product (n:ns) =
```

Step 3 - Define the simple case

By definition, the product of zero integers is one, because one is the identity for multiplication.

So:

```
product [] = 1
product (n:ns) =
```

Step 4 - Define other cases

Consider the ingredients that can be used, e.g.

- ☐ the function itself (product) and
- ☐ the arguments (n and ns) and
- □ library functions of the relevant types (+, -, *, /)

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

The other cases are often recursive cases.

Step 5 - Generalise and Simplify

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

We can generalise from integers to any numeric type

```
product :: Num a => [a] -> a
product [] = 1
product (n:ns) = n * product ns
```

We will see how to simplify the definition later

Example 2 - drop using steps

drop – the function that takes an integer (n) and a list of values of some type a and returns a list with the first n elements dropped.

Step 1 - Define the type

(Use the simplest type possible)

Step 2 - Enumerate the cases

For lists, these are

- ☐ the empty list and
- ☐ the non-empty list

```
drop 0 [] =
drop 0 (x:xs) =
drop n [] =
drop n (x:xs) =
```

Step 3 - Define the simple case

- ☐ By definition, removing zero elements from the start of a list gives the same list gives the same list.
- □ Removing n elements from an empty list can return an empty list (for safety).

So:

```
drop 0 [] = []
drop 0 (x:xs) = x:xs
drop n [] = []
drop n (x:xs) =
```

Step 4 - Define other cases

Consider the ingredients that can be used, e.g.

- ☐ the function itself (drop) and
- ☐ the arguments (x and xs) and
- □ library functions of the relevant types (+, -, *, /)

```
drop 0 [] = []
drop 0 (x:xs) = x:xs
drop n [] = []
drop n (x:xs) = drop (n-1) xs
```

Step 5 - Generalise and Simplify

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

We can generalise from integers to any integral type

```
drop :: Integral b => b -> [a] -> [a]
drop 0 [] = []
drop 0 (x:xs) = x:xs
drop n [] = []
drop n (x:xs) = drop (n-1) xs
```