Added list comprehensions and recursive functions
hamayumm authored 4 days ago

b48ee3ea

recursive_functions.md 11.3 KB

Recursive Functions

These notes should be read in conjunction with chapter 6 of our textbook Programming in Haskell.

- We discuss some examples from the Haskell'98 standard prelude for pedagogical purposes.
- See the prelude for the current version of the language for all predefined classes and their instances.

Basic Concepts

There is also a video on this section.

As we have seen, many functions can naturally be defined in terms of other functions. For example, a function that returns the factorial of a non-negative integer can be defined by using library functions to calculate the product of the integers between one and the given number:

```
fac :: Int -> Int
fac n = product [1..n]
```

Expressions are evaluated by a stepwise process of applying functions to their arguments.

For example:

```
fac 5
= product [1..5]
= product [1,2,3,4,5]
= 1*2*3*4*5
= 120
```

In Haskell, functions can also be defined in terms of themselves. Such functions are called **recursive**.

```
fac :: Int -> Int
fac 0 = 1
fac n = n * fac (n-1)
```

The function **fac** maps 0 to 1 (*base case*), and any other integer to the product of itself and the factorial of its predecessor (*recursive case*).

For example:

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

Note:

- fac 0 = 1 is appropriate because 1 is the identity for multiplication: 1 * x = x = x * 1.
- The recursive definition diverges on integers < 0 because the base case is never reached:

```
> fac (-1)
*** Exception: stack overflow
```

Why is Recursion Useful?

- Some functions, such as factorial, are simpler 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 *induction*.

Recursion on Lists

There is also a video on this section.

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

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

The `product'` function maps the empty list to 1 (*base case*), and any non-empty list to its head multiplied by the `product'` of its tail (*recursive case*).

For example:

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

Note: Lists in Haskell are actually constructed one element at a time using the **cons** operator. Hence, [2,3,4] is just an abbreviation for 2:(3:(4:[])).

Using the same pattern of recursion as in `product'` we can define the length function on lists.

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

The length function maps the empty list to 0 (*base case*), and any non-empty list to the successor of the length of its tail (*recursive case*).

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]
```

The function 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]
```

Interestingly, the append opearator ++ used in the above example can also be defined using recursion.

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

For example:

```
[1,2,3] ++ [4,5]
= 1 : ([2,3] ++ [4,5])
= 1 : (2 : ([3] ++ [4,5]))
= 1 : (2 : (3 : ([] ++ [4,5])))
= 1 : (2 : (3 : [4,5]))
= [1,2,3,4,5]))
```

The recursive definition of ++ formalises the idea that two lists can be appended by copying elements from the first list until it is exhausted, at which point the second list is joined-on at the end.

Multiple Arguments

There is also a video on this section.

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

- Zipping the elements of two lists:

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

For example:

```
zip ['a', 'b', 'c'] [1,2,3,4]
= ('a',1) : zip ['b', 'c'] [2,3,4]
= ('a',1) : ('b',2) : zip ['c'] [3,4]
= ('a',1) : ('b',2) : ('c',3) : zip [] [4]
= ('a',1) : ('b',2) : ('c',3) : []
= [('a',1), ('b',2), ('c',3)]
```

- Remove the first n elements from a list:

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

For example:

```
drop 3 [4,6,8,10,12]
= drop 2 [6,8,10,12]
= drop 1 [8,10,12]
= drop 0 [10,12]
[10,12]
```

Multiple Recursion

There is also a video on this section.

Functions can also be defined using *mulitple recursion*, in which a function is applied more than once in its own definition.

A function that calculates the nth Fibonacci number (0, 1, 1, 2, 3, 5, 8, 13, ...) for any integer n >= 0 can be defined using double recursion as follows:

```
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n-2) + fib (n-1)
```

For example:

```
fib 4
= fib (2) + fib (3)
= fib (0) + fib (1) + fib (1) + fib (2)
= 0 + 1 + 1 + fib (0) + fib (1)
= 0 + 1 + 1 + 0 + 1
```

```
= 3
```

Mutual Recursion

There is also a video on this section.

Functions can also be defined using *mutual recursion*, where two or more functions are defined recursively in terms of each other. For non-negative integers, we can define even and odd numbers using mutual recursion.

```
even :: Int -> Bool
even 0 = True
even n = odd (n-1)

odd :: Int -> Bool
odd 0 = False
odd n = even (n-1)
```

For example:

```
even 4
= odd 3
= even 2
= odd 1
= even 0
= True
```

Similarly, functions that select the elements from a list at all even and odd positions (counting from zero) can be defined as follows:

```
evens :: [a] -> [a]
evens []     = []
evens (x:xs) = x : odds xs

odds :: [a] -> [a]
odds []     = []
odds (_:xs) = evens xs
```

For example:

```
evens "abcde"
= 'a' : odds "bcde"
= 'a' : evens "cde"
= 'a' : 'c' : odds "de"
= 'a' : 'c' : evens "e"
= 'a' : 'c' : 'e' : odds []
= 'a' : 'c' : 'e' : []
= "ace"
```

Programming Example - Quicksort

There is also a video on this section.

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

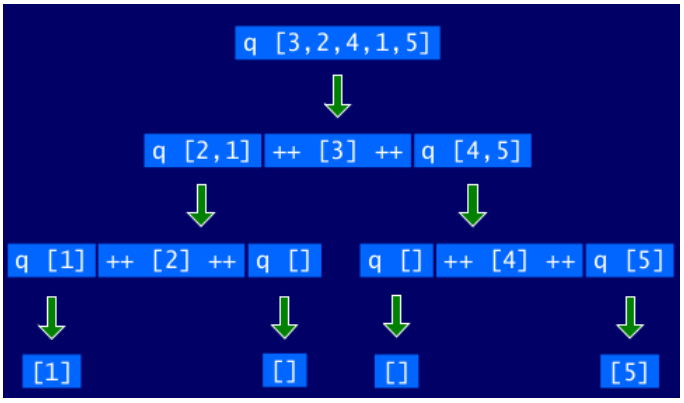
- The empty list is already sorted;
- Non-empty lists can be sorted by sorting the tail values <= the head, sorting the tail values > the head, and then appending the resulting lists on either side of the head value.

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

```
qsort :: Ord a => [a] -> [a]
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 simplest implementation of quicksort in any programming language!

For example (abbreviating qsort as q):



Advice on Recursion

There is also a video on this section.

In this section, we offer some advice for defining functions in general, and recursive functions in particular, using a five-step process.

Example - drop: that removes a given number of elements from the start a list.

- Step 1: define the type

The drop function takes an integer and a list of values of some type a, and produces another list of such values.

```
drop :: Int -> [a] -> [a]
```

We have already made four design decisions in defining this type:

(i) using integers rather than a more general numeric type - for simiplicity

(ii) using currying rather than taking arguments as a pair - for flexibility

(iii) supplying the integer argument before the list argument - for readability (*drop* **n** *elements from* **xs**)

(iv) making the function *polymorphic* in the type of the list elements - for generality.

- Step 2: enumerate the cases

We have a total of four possible cases i.e. two possible values for integer argument (0 and n) and two possibilities for the list argument ([] and (x:xs)) thus giving us four possible combinations (cases).

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

- Step 3: define the simple cases

By definition, removing zero elements from the start of any list gives the same list.

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

Attempting to remove one or more elements from the empty list is invalid, so the third case could be omitted, which would result in an error being produced if this situation arises. However, we choose to avoid the production of an error by returning the empty list in this case:

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

- Step 4: define the other cases

For removing one or more elements from a non-empty list, we drop the head of the list and recursively call itself with one less than the previous call on

the tail of the list.

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

- Step 5: generalise and simplify

The first two equations for drop can be combined into a single equation that states that removing zero elements from any list gives the same list:

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

The variable n in the second equation and x in the third can be replaced by the wildcard pattern _, as these variables are not being used in the bodies of their equations.

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

This is precisely the same definition of drop function as available in the standard prelude.

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 :: Ord a => [a] -> [a] -> [a]
```

that merges two sorted lists of values to give a single sorted list.

For example:

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