Chapter 6

Recursive Functions

Connor Baker

Compiled on December 21, 2019 at 9:04pm

6.1 Basic Concepts

Functions can be defined in terms of themselves. When a function is self-referential, we say that it is recursive. Consider the explicit and recursive definitions of the factorial function:

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

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

The first type of case in a recursive definition is called the base case (there can be more than one base case in a recursive definition). The second type of case is called the recursive case, which covers all other cases besides the base case.

```
fac 3
={ applying fac }
3 * fac 2
={ applying fac }
3 * (2 * fac 1)
={ applying fac }
3 * (2 * (1 * fac 0))
={ applying fac }
3 * (2 * (1 * 1))
={ applying * }
6
```

> Graham Hutton. “Programming in Haskell” (2nd ed.).

Though explicit function definitions are typically more performant, they can also be more difficult to write. Recursive definitions give programmers the ability to express certain properties of functions in an elegant and concise way. Consider the following definition of multiplication for non-negative integers.

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

> Graham Hutton. “Programming in Haskell” (2nd ed.).

6.2 Recursion on lists

Recursion can be used to easily define functions that operate on lists. Consider the product function which can be defined to operate on a list.

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

> Graham Hutton. “Programming in Haskell” (2nd ed.).

Remember that lists in Haskell are built one element at a time using the `cons` operator. We can think of `[1,2,3]` as syntactic sugar for `1:(2:(3:[]))`. With this understanding, we can think of a recursive definition for calculating the length of list:

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

> Graham Hutton. “Programming in Haskell” (2nd ed.).

Note the use of the wildcard pattern `_` in the recursive case, which reflects the fact that calculating the length of a list does not depend upon the values of its elements.

Similarly, we can define a function that reverses the list:

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

> Graham Hutton. “Programming in Haskell” (2nd ed.).

Interestingly enough, we can define the append operator, `++`, as

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

> Graham Hutton. “Programming in Haskell” (2nd ed.).

We can also define a function that inserts an element into a list, as long as the elements are comparable.

```
insert :: Ord a => a -> [a] -> [a]
insert x []                 = [x]
insert x (y:ys) | x <= y    = x : y : ys
                | otherwise = y : insert x ys
```

> Graham Hutton. “Programming in Haskell” (2nd ed.).

In particular, if `x <= y` then the new element `x` is simply prepended to the start of the list, otherwise the head `y` becomes the first element of the resulting list, and we then proceed to insert the new element into the tail of the given list. For example, we have:

```
insert 3 [1,2,4,5]
={ applying insert }
1 : insert 3 [2,4,5]
={ applying insert }
1 : 2 : insert 3 [4,5]
```

```
={ applying insert }
1 : 2 : 3 : [4,5]
={ list notation }
[1,2,3,4,5]
```

Graham Hutton. “Programming in Haskell” (2nd ed.).

With insert done, we can write a function that sorts an an existing list. It works be inserting the head of the list into itself, repeatedly sorting its tail.

```
isort :: Ord a => [a] -> [a]
isort []= []
isort (x:xs) = insert x (isort xs)
```

For example:

```
isort [3,2,1,4]
={ applying isort }
insert 3 (insert 2 (insert 1 (insert 4 [])))
={ applying insert }
insert 3 (insert 2 (insert 1 [4]))
={ applying insert }
insert 3 (insert 2 [1,4])
={ applying insert }
insert 3 [1,2,4]
={ applying insert }
[1,2,3,4]
```

Graham Hutton. “Programming in Haskell” (2nd ed.).

6.3 Multiple Arguments

Recursion is not limited to functions of a single argument. Consider the function `zip`, which stitches two lists together.

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

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

We can also define a function which removes some number of elements from the front of a list:

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

Graham Hutton. “Programming in Haskell” (2nd ed.).

6.4 Multiple Recursion

Recursive functions can invoke themselves more than once; consider the following definition of the n^{th} Fibonacci number:

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

Graham Hutton. “Programming in Haskell” (2nd ed.).

Now that we understand recursion, we should have a newfound appreciation for the definition of quick sort:

```
qsort :: Ord a => [a] -> [a]
qsort []    = []
qsort (x:s) = qsort smaller ++ [x] ++ qsort larger
              where
                    smaller = [a | a   x]
```

Graham Hutton. “Programming in Haskell” (2nd ed.).

6.5 Mutual Recursion

Two or more functions are called mutually recursive when they are defined in terms of each other. Consider the following definition of odd and even, for non-negative integers:

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

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

> Graham Hutton. “Programming in Haskell” (2nd ed.).

Further functions can be defined in terms of these two that operate on lists, yielding elements at only odd or even indices:

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

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

> Graham Hutton. “Programming in Haskell” (2nd ed.).

6.6 Advice on Recursion

Hutton covers what he describes as a “five-step process” for defining recursive functions. The steps are as follows:

- Define the type of the function
- Though the compiler can infer the types, it’s best to make sure that you understand what the function you wrote is doing
- Enumerate the cases
- List all of the cases that could possibly occur
- Define the simple cases
- Take care of the base case, or some of the simpler recursive cases
- Define the other cases
- Tackle the more difficult recursive cases
- Generalize and simplify
- Look back on your solution and find ways to improve it – do some of the cases reduce to others that you’ve already written?

6.7 Chapter Remarks

Hutton notes that the five-step process he describes for creating recursive functions is based on Glaser, Hartel, and Garratt’s “Programming by Numbers: A Programming Method for Novices.” It can be found in The Computer Journal, vol. 43, no. 4, (published in 2000).