Functional Programming

Paul Mellies

Lesson 8: Writing Recursive Functions
The general recipe for writing recursive functions

We have seen in the previous lesson that one may follow these guidelines in order to write a recursive function in Haskell:

1. Identify the end goal(s) of the problem.
2. Determine what happens when a goal is reached.
3. List all alternate possibilities.
4. Determine your "rinse and repeat" process.
5. Ensure that each alternative brings you closer to the goal.

We carry on our exploration and explain how to use pattern-matching in order to resolve problems recursively in an elegant and easy way.
Recursion on lists: the \texttt{myLength} function

The \texttt{myLength} function computes the length of a list by recursion:

\begin{verbatim}
myLength [] = 0
myLength xs = 1 + myLength (tail xs)
\end{verbatim}

\textbf{Exercise:} use pattern-matching to define the \texttt{myLength} function without explicitly calling the \texttt{tail} function.
Recursion on lists: the **myLength** function

The **myLength** function computes the length of a list by recursion:

\[
\text{myLength } [] = 0 \\
\text{myLength } \text{xs} = 1 + \text{myLength } \text{(tail } \text{xs)}
\]

**Exercise:** use pattern-matching to define the **myLength** function without explicitly calling the **tail** function.

**Answer:**

\[
\text{myLength } [] = 0 \\
\text{myLength } (x:xs) = 1 + \text{myLength } xs
\]
Recursion on lists: the myTake function

The `take` function has two arguments: an integer `n` and a list `list`. The purpose of the function is to compute the list of the `n` first elements of `list`.

In contrast to `tail` and `head`, the function has no problem with the empty list, and will return as many items as it can.

In particular, applying the function on the empty list always returns the empty list. Moreover, applying the function to `0` and any list always returns the empty list.

This leads to the following end goals for the `myTake` function:

```
myTake _ [] = []
myTake 0 _ = []
```
Recursion on lists: the myTake function

In the case of our length function, we had to worry only about taking apart the list, while in the case of the myTake function, we also have to build the list we return.

Think of the following function call:

\[
\text{myTake 3 [1,2,3,4,5]}
\]

This is the way you want the function to behave in this case:

1. pick the first element 1 and cons it along with take 2 [2,3,4,5]
2. pick the next element 2 and cons it along with take 1 [3,4,5]
3. then pick the element 3 and cons it along with take 0 [4,5]
4. at 0 the function has reached its end goal, so return the empty list []
5. this produces 1:2:3:[] which is precisely the list [1:2:3]
Recursion on lists: the \texttt{myTake} function

The algorithm can be nicely implemented in Haskell using pattern matching:

\begin{verbatim}
myTake n (x:xs) = x:rest
  where rest = myTake (n-1) xs
\end{verbatim}

Putting all together, one obtains the following Haskell code for the \texttt{myTake} function:

\begin{verbatim}
myTake _ [] = []
myTake 0 _ = []
myTake n (x:xs) = x:rest
  where rest = myTake (n-1) xs
\end{verbatim}

Note that the \texttt{myTake} function may terminate in two quite different situations: (1) when the integer \texttt{n} gets to zero or (2) when the list under scrutiny is empty.
Recursion on lists: the \texttt{myCycle} function

The \texttt{cycle} function is a particularly exciting function to encode in Haskell.

This is also a function which works thanks to the lazy evaluation process of Haskell, and which may be thus difficult to write in another language.

The function takes a list as input and repeats it for ever.

The function may be written as follows in Haskell:

\[
\texttt{myCycle (first:rest) = first : myCycle (rest ++ \{first\})}
\]

Note that, quite remarkably, the function has no end goal state.
The Ackermann function

Remember that the Ackermann function is the total recursive function

\[ A : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \]

defined by the three rules:

\[
\begin{align*}
A(0, n) &= n + 1 \\
A(m, 0) &= A(m - 1, 1) \\
A(m, n) &= A(m - 1, A(m, n - 1))
\end{align*}
\]

where the second rule applies precisely when \( m \geq 1 \) and \( n = 0 \),

and the third rule applies precisely when \( m \geq 1 \) and \( n \geq 1 \).
The Ackermann function

The Ackermann function may be nicely implemented in Haskell:

```haskell
ackermann 0 n = n + 1
ackermann m 0 = ackermann (m-1) 1
ackermann m n = ackermann (m-1) (ackermann m (n-1))
```

Note that each recursive call is making nested calls to itself!

Indeed, we have seen in the lesson on recursive functions that the Ackermann function is total recursive but not primitive recursive!
The Ackermann function

One nice feature of the interactive mode of Haskell is the command

```
:set +s
```

which enables one to time each function call in ghci.

The Ackermann function then behaves as follows:

```
ghci> :set +s
ghci> ackermann 3 3
61
(0.01 secs)
ghci> ackermann 3 8
2045
(3.15 secs)
ghci> ackermann 3 9
4093
(12.97 secs)
```
The Collatz conjecture

Consider a strictly positive natural number \( n \in \mathbb{N} \) and apply the following procedure:

- If the number \( n \) is even, then divide \( n \) by 2. Repeat.
- If the number \( n \) is odd, then multiply \( n \) by 3 and add 1. Repeat.
- Stop the procedure when the number \( n \) is equal to 1.

The Collatz conjecture is that:

the procedure always terminates on the number 1.
The Collatz conjecture

In other words...

The Collatz conjecture states that the following Haskell program always terminates:

```
collatz 1 = 1
collatz n = if even n
    then 1 + collatz (n `div` 2)
    else 1 + collatz (n*3 + 1)
```

A very difficult conjecture to resolve...

The famous mathematician Paul Erdős even said:

« Mathematics may not be ready for such problems »
The Collatz conjecture

We can test on the machine how many steps it takes to reach the value 1:

```ghci
ghci> collatz 9
20
ghci> collatz 999
50
ghci> collatz 92
18
ghci> map collatz [100 .. 120]
[26,26,26,88,13,39,13,101,114,114,114,70,21,13,34,34,21,21,34,34,21]
```
The Collatz conjecture

We can test on the machine how many steps it takes to reach the value 1:

```
ghci> collatz 9
20
ghci> collatz 999
50
ghci> collatz 92
18
ghci> map collatz [100 .. 120]
[26,26,26,88,13,39,13,101,114,114,114,114,70,21,13,34,34,21,21,34,34,21]
```
The Collatz conjecture

The \texttt{collatz} function may be decomposed and rewritten as

\begin{verbatim}
collatz 1 = 1
collatz n = 1 + collatz (collatzstep n)
\end{verbatim}

where the \texttt{collatzstep} function is defined as follows:

\begin{verbatim}
collatzstep 1 = 1
collatzstep n = if even n then n 'div' 2
             else n*3 + 1
\end{verbatim}
The Collatz conjecture

Instead of counting the number of steps to reach 1 from the natural number \( n \), we can track the list of all natural numbers from \( n \) using the \texttt{collatzlist} function:

\[
\text{collatzlist } 1 = [1] \\
\text{collatzlist } n = n : \text{collatzlist} (\text{collatzstep } n)
\]

where the \texttt{collatzstep} function is defined as follows:

\[
\text{collatzstep } 1 = 1 \\
\text{collatzstep } n = \text{if } \text{even } n \\
\quad \text{then } n \mod 2 \\
\quad \text{else } n \times 3 + 1
\]

Note that \texttt{collatz } \( n \) is the length of the list \texttt{collatzlist } \( n \).
Thank you for your attention!