Recursion in Haskell

COMP2209 - Programming III

Dr Julian Rathke

These slides use modified content from Graham Hutton’s slides for the module textbook Programming in Haskell
Recursion, the story so far

• We learned in Programming I and II that we can program methods in Java using **recursion**.

• That is, a method may contain a call to itself within its own definition.

• We also learned that doing so in Java has to be done with care to avoid StackOverflow Exceptions.

• Every recursive call in Java pushes a new frame on to the call stack.

• We learned that recursively defined methods in Java can be very useful in certain cases for solving problems elegantly but are not necessarily the most efficient.

• cf. iterative vs. recursive definition of Fibonacci numbers.
Recursion in Haskell

- It will be no surprise that you can define functions recursively in Haskell also simply by using function application of a function name in the body of its definition.

\[
\begin{align*}
\text{fac } 0 &= 1 \\
\text{fac } n &= n \times \text{fac } (n-1)
\end{align*}
\]

- To understand the execution of \text{fac } n, say, we can simply rewrite the LHS of an equation with the RHS:

\[
\begin{align*}
\text{fac } 3 &= 3 \times \text{fac } 2 \\
&= 3 \times (2 \times \text{fac } 1) \\
&= 3 \times (2 \times (1 \times \text{fac } 0)) \\
&= 3 \times (2 \times (1 \times 1)) \\
&= 3 \times (2 \times 1) \\
&= 6
\end{align*}
\]

\[
\begin{align*}
\text{fac } -1 &= -1 \times \text{fac } -2 \\
&= -1 \times (-2 \times \text{fac } -3) \\
&= -1 \times (-2 \times (-3 \times \text{fac } -4)) \\
&\quad \ldots
\end{align*}
\]

*** Exception: stack overflow
Recursion over structured data

• You can see from the previous example that recursive functions may not always terminate when they are called.

• A useful way of guaranteeing termination is to ensure that the recursion is defined over a structured data type and that each recursive call is made to a smaller structure than the current call.

• For example, for recursion over lists

\[
\text{product} :: \text{Num} \Rightarrow [a] \rightarrow a
\]

\[
\text{product} [] = 1
\]

\[
\text{product} (x:xs) = x * \text{product} \ xs
\]

• The recursive call to \texttt{product} is on a smaller structure than the original argument, this guarantees that the recursion will terminate whatever (finite) list is given.
A common recursion pattern

- We can use the same basic pattern for recursion over lists to define many functions.

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

- Consider how this is evaluated?
- Notice the growing calculation of
  - \(1 + 1 + 1 \ldots\) etc

```haskell
length [1,2,3]
= 1 + length [2, 3]
= 1 + (1 + length [3])
= 1 + (1 + (1 + length []))
= 1 + (1 + (1 + 0))
= 1 + (1 + 1)
= 1 + 2
= 3
```
And again, building a list

\[
\text{reverse :: } [a] \rightarrow [a] \\
\text{reverse [] } = [\] \\
\text{reverse (x:xs)} = \text{reverse xs } ++ [x]
\]

\[
\text{reverse [1,2,3]} \\
= \text{reverse [2,3] } ++ [1] \\
= (((\text{reverse [] } ++ [3]) + + [2]) ++ [1] \\
= ([[] ++ [3]) + + [2]) ++ [1] \\
= ([3] ++ [2]) ++ [1] \\
= [3,2,1]
\]

Notice the growing calculation again. We’ll come back to that.
Recursion on Multiple Arguments

- We can see that recursion is similar to list comprehension in that it lets us iterate over data structures.
- It is more flexible in that it lets us iterate over multiple structures simultaneously.
- This is the basis of the `zip` function we studied already.

```haskell
zip [] _ = []
zip _ [] = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

- This is a general idea that can be used with different types also

```haskell
drop 0 xs = xs
drop _ [] = []
drop n (_:xs) = drop (n-1) xs
```
Mutual Recursion

- Haskell supports definition of functions using **mutual recursion** in which two or more functions are defined in terms of each other.
- Let’s define functions that select the odd and even characters from a string (counting from 0).

\[
\begin{align*}
\text{evens} :: [a] & \rightarrow [a] \\
\text{evens} \; [] & = [] \\
\text{evens} \; (x:xs) & = x : \text{odds} \; xs
\end{align*}
\]

\[
\begin{align*}
\text{odds} :: [a] & \rightarrow [a] \\
\text{odds} \; [] & = [] \\
\text{odds} \; (x:xs) & = \text{evens} \; xs
\end{align*}
\]

\[
\begin{align*}
\text{evens} \; \text{"abcde"} & = \text{‘a’} : \text{odds} \; \text{"bcde"} \\
& = \text{‘a’} : \text{evens} \; \text{"cde"} \\
& = \text{‘a’} : \text{‘c’} : \text{odds} \; \text{"de"} \\
& = \text{‘a’} : \text{‘c’} : \text{evens} \; \text{“e”} \\
& = \text{‘a’} : \text{‘c’} : \text{‘e’} : \text{odds} \; [] \\
& = \text{‘a’} : \text{‘c’} : \text{‘e’} : [] \\
& = \text{“ace”}
\end{align*}
\]
5 Steps to Better Recursion

• The next few slides will detail advice about how to go about writing your own recursive functions. It mostly just takes practice.

• These steps are from Glaser, Hartel and Garratt “Programming by Numbers: A Programming Method for Novices”.

Step 1: Define the Type
Step 2: Enumerate the Cases
Step 3: Define the simple (base) cases
Step 4: Define the other (inductive) cases
Step 5: Generalise and Simplify

These are best understood by example.
product in 5 steps

Step 1 - Define the Types

\[ \text{product} :: [\text{Int}] \to \text{Int} \]

Step 2 - List the cases

\[ \text{product } [] = ??? \]
\[ \text{product } (n:ns) = ??? \]

Step 3 - Simple Cases

\[ \text{product } [] = 1 \]
\[ \text{product } (n:ns) = ??? \]

This function takes a list of Int and returns the product (an Int) of them.

The cases here are the empty list and non-empty list.

The unit for multiplication is 1.
product in 5 steps

Step 4 - Other cases

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

Hardest bit: consider what can be used. The recursive call gives the product of the rest of the list.

Step 5 - Generalise and Simplify

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

There is no need to restrict this function to Int types as it works on any type that supports *
Step 1 - Define the Types

\texttt{init :: [a] \rightarrow [a]}

This function takes a list and removes the last element from it.

Step 2 - List the cases

\texttt{init [] = ???}
\texttt{init (x:xs) = ???}

The cases here are the empty list and non-empty list. But empty list is not a valid input here.

Step 3 - Simple Cases

\texttt{init (x:xs) | null xs = []}
| otherwise = ???

Needs a bit of thought here to spot whether \texttt{x} is the last element. \texttt{null} checks for empty list.
init in 5 steps

**Step 4 - Other cases**

```
init (x:xs) | null xs = []
| otherwise = x : init xs
```

To remove the last element from a list of at least two elements. Keep the head and remove the last element from the tail.

**Step 5 - Generalise and Simplify**

```
init :: [a] → [a]
init [] = []
init (x:xs) = x : init xs
```

The guards only look at the structure of xs so we can use pattern matching instead. x is thrown away so a wildcard is used.
Let’s talk about tail calls.

• If you read about recursive functions in various languages you will soon encounter discussions about tail recursion and tail call optimisation.

• A recursive function is tail recursive if, put very roughly, there is no suspended computation built up in the execution stack as the recursion unfolds.

• For example

\[
\begin{align*}
\text{fac} & : : \text{Int} \rightarrow \text{Int} \\
\text{fac } 0 & = 1 \\
\text{fac } n & = n \times \text{fac} \ (n-1)
\end{align*}
\]

• Is **not** tail recursive!

\[
\begin{align*}
\text{fac } 3 & \\
& = 3 \times \text{fac } 2 \\
& = 3 \times (2 \times \text{fac } 1) \\
& = 3 \times (2 \times (1 \times \text{fac } 0)) \\
& = 3 \times (2 \times (1 \times 1)) \\
& = 3 \times (2 \times 1) \\
& = 3 \times 2 \\
& = 6
\end{align*}
\]
A tail recursive function

- Let’s rewrite the `fac` function to make it *look* tail recursive

```haskell
fac' :: Int → Int → Int
fac' acc 0 = acc
fac' acc n = fac' (n*acc) (n-1)
```

Example:

```haskell
fac' 3 1
= fac’ (3*1) (3-1)
= fac’ 3 2
= fac’ (3*2) (2-1)
= fac’ 6 1
= fac’ (6*1) (1-1)
= fac’ 6 0
= 6
```

We might like to think that this implementation is better as it will not grow the multiplication as the recursion unfolds.

This is NOT TRUE in Haskell!
A tail recursive function - a closer look

\[
\text{fac'} \ 3 \ 1 \\
= \text{fac'} \ (3*1) \ (3-1) \\
= \text{fac'} \ 3 \ 2 \\
= \text{fac'} \ (3*2) \ (2-1) \\
= \text{fac'} \ 6 \ 1 \\
= \text{fac'} \ (6*1) \ (1-1) \\
= \text{fac'} \ 6 \ 0 \\
= 6
\]

Haskell is a lazy language - it evaluates only the expressions it needs to.

It doesn't need to know what 3*1 is in order to rewrite the call to fac'.

It does need to calculate 3-1 so that it can pattern match the definition of fac'.
A tail recursive function - actual steps

fac’ :: Int → Int → Int
fac’ acc 0 = acc
fac’ acc n = fac’ (n*acc) (n-1)

fac’ 3 1
= fac’ (3*1) (3-1)
= fac’ (3*1) 2
= fac’ (3*1*2) (2-1)
= fac’ (3*1*2) 1
= fac’ (3*1*2*1) (1-1)
= fac’ (3*1*2*1) 0
= 6

Thus, a tail recursive version of this function does not gain much efficiency in Haskell.

You can force evaluation of the growing expression though - more on that in a later lecture.
Tail Recursion - efficiency gain

- Tail recursive functions in Haskell can sometimes produce efficiency gains.
- Consider the reverse function on lists.

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

This is clearly not tail recursive

There are numerous append operations pending.
Tail recursive reverse

reverse' :: [a] → [a] → [a]
reverse' [] acc = acc
reverse' (x:xs) acc = reverse' xs (x:acc)

reverse' [1,2,3] []
= reverse' [2,3] (1:[])
= reverse' [3] (2:1:[])
= reverse' [] (3:2:1:[])
= (3:2:1:[])
= [3,2,1]

The resulting list is already constructed and the pending ++ operations have gone
Tail Recursion - efficiency loss

• Be careful though, tail recursion doesn’t always produce better results.
• For recursion where the output is structured data and the recursive call is guarded by the value constructor then Haskell’s laziness let’s us use non-tail recursive calls effectively.
• For example: consider the function that replicates a given value a given number of times in a list.
• We’ll define this both directly and tail recursively to see the difference.
• Note that the tail recursively version here suffers terrible when only a portion of the output is required.
replicate - tail vs non-tail

replicate :: Int → a → [a]
replicate 0 x = []
replicate n x = x : replicate (n-1) x

This is a guarded recursive call

replicate' :: Int → a → [a] → [a]
replicate' 0 x acc = acc
replicate' n x acc = replicate (n-1) x (x:acc)

Tail Recursive
Replicate - tail vs non-tail

Consider:  
\[
\text{take 2 (replicate 1000000 'a')}
\]

\[
\begin{align*}
\text{take 2 (replicate 1000000 'a')} & \\
& = \text{take 2 ('a': replicate 999999 'a')} \\
& = 'a': \text{take 1 (replicate 999999 'a')} \\
& = 'a': 'a': \text{take 0 (replicate 999998 'a')} \\
& = 'a': 'a': [] \\
& = "aa"
\end{align*}
\]

Vs

\[
\text{take 2 (replicate' 1000000 'a' [])}
\]

\[
\begin{align*}
\text{take 2 (replicate' 1000000 'a' [])} & \\
& = \text{take 2 (replicate 999999 'a' ['a'])} \\
& = \text{take 2 (replicate 999998 'a' ['a','a'])} \\
& = \text{take 2 (replicate 999997 'a' ['a','a','a'])} \\
& = \text{take 2 (replicate 999996 'a' ['a','a','a','a'])} \\
& = \ldots
\end{align*}
\]
YOUR QUESTIONS

Next Lecture:
Higher-Order Functions