Unit 4c: Recursion & Optimisation

1 Introduction

Previously we noted that recursion can cause stack overflow as each recursive call creates a new stack frame. Most functional languages have the capability to optimise recursive calls to avoid this problem. They do so because recursion is an important technique for programming in a functional style.

To understand how to make use of this, we need to distinguish between:

1.1 Head recursion

A function makes its recursive call and then performs some more calculations, maybe using the result of the recursive call.

1.2 Tail recursion

All calculations happen first and the recursive call is the last thing that happens.

1.3 Factorial Example

So which of these is this factorial example (similar to the one you saw previously)?

```
def factorial(n: Int): Int = {
  if (n<=1)
     n
  else
     factorial(n-1) * n
}
```

Although the recursive call to factorial is in the last line of the function, the call is not the last act in evaluating the expression in that line.

To evaluate this:

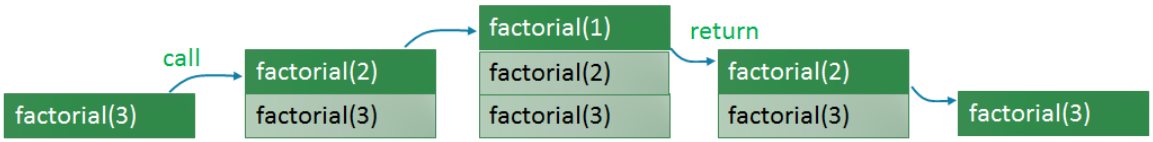
1. Function is called and value of *factorial(n-1)* returned
2. That value is multiplied by *n*

So the multiplication calculation is done *after* the recursive call – *head recursion*.

2 Recursion and the stack frame

What happens on the stack if we call factorial(3)?

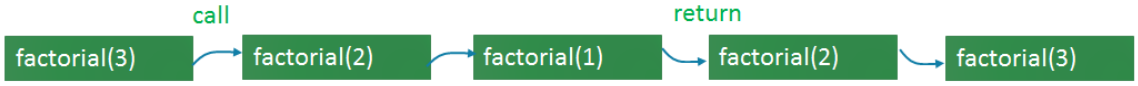
2.1 Non-optimised call



Each box represents a *stack frame*. Stack memory is allocated to store function (or method) call with local variables/parameters and removed from stack only when function execution completes. Each recursive call creates a *new frame* – all these stack frames are separate calls to the same function.

2.2 Optimising recursion

Optimisation works by re-using the stack frame for the next function call. This is only possible if all other work (e.g. calculations) in the function is completed before recursive call, so local variables/parameters in that frame are no longer needed – needs *tail recursion*.



The same frame is re-used for each call; the stack use doesn't "grow" so will not overflow. Scala compiler does this optimisation "under the hood" when it identifies a tail recursive call.

Note that recursion is not exclusively a feature of functional languages, but is more likely to be used by "functional-thinking" programmers so compilers of functional languages are more likely to implement this optimization.

3 Making functions tail-recursive

To take advantage of optimisation we need to modify the *factorial* function to be tail recursive. A common technique for situations like this is to use a *nested function* as a helper:

```
def factorial (i: Int): Int = {

  def fact(i: Int, accumulator: Int): Int = {
    if (i <= 1)
      accumulator
    else
      fact(i - 1, i * accumulator)
  }

  fact(i, 1)
}
```

Note the *factorial* function has same signature as before; inside this factorial method we **declare** and **call** the nested function *fact*. *fact* function is tail recursive – no calculation done after the function call returns – multiplication is done before function call and result passed as parameter.

Can you follow how the fact method works? Why is the second parameter value in the nested call 1? Hint: it essentially does the calculation in the reverse order to the original version of factorial.

4 Why use a nested function for tail recursion?

If you look at the nested method *fact* in the example, it actually does all the work of calculating the factorial. You could have *fact* as a standalone function, and call it, e.g. *fact(4,1)*, which would calculate the factorial of 4 correctly. There are two problems with this:

1. The second parameter is only there because of the way the function works, it has no meaning in the context of the purpose of the function – you just want to calculate factorial of 4
2. There would be nothing to prevent a call such as *fact(4,2)*. What would this mean? Well, it wouldn't calculate the factorial of 4, anyway.

Nesting inside the *factorial* method avoids these problems:

1. Provides method with sensible signature, e.g. call *factorial(4)* to get the factorial of 4 – simples!
2. Prevents fact being called directly with possibly meaningless results – scope of method is inside the method where it was declared