Extracted from:

Pragmatic Scala

Create Expressive, Concise, and Scalable Applications

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Scala 2.11

Pragmatic Scala

Create Expressive, Concise, and Scalable Applications

Venkat Subramaniam

edited by Jacquelyn Carter

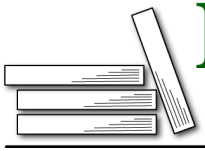
Pragmatic Scala

Create Expressive, Concise, and Scalable Applications

Venkat Subramaniam

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *https://pragprog.com*.

The team that produced this book includes:

Jacquelyn Carter (editor)
Potomac Indexing, LLC (index)
Liz Welch (copyedit)
Dave Thomas (layout)
Janet Furlow (producer)
Ellie Callahan (support)

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-054-7
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—September 2015

CHAPTER 11

Programming Recursions

The idea of recursion—solving a problem using solutions to its subproblems—is alluring. Many algorithms and problems are recursive in nature. Once we get the hang of it, designing solutions using recursion becomes highly expressive and intuitive.

In general, the biggest catch with recursions is stack overflow for large input values. But, thankfully that's not so in Scala for specially structured recursions. In this chapter we explore the powerful tail call optimization techniques and the support classes baked into Scala and its library, respectively. Using these easy-to-access facilities, you can implement highly recursive algorithms and reap their benefits for really large input values without blowing out the stack.

A Simple Recursion

Recursion is used quite extensively in a number of algorithms, like quick sort, dynamic programming, stack-based operations…and the list goes on. Recursion is highly expressive and intuitive. Sometimes we also use recursion to avoid mutation. Let's look at a use of recursion here. We'll keep the problem simple so we can focus on the issues with recursion instead of dealing with problem or domain complexities.

ProgrammingRecursions/factorial.scala

```
Line 1 def factorial(number: Int) : BigInt = {
   2   if(number == 0)
   3     1
   4   else
   5     number * factorial(number - 1)
   6 }
```

The factorial() function receives a parameter and returns a value of 1 if the parameter is zero. Otherwise, it recursively calls itself to return a product of that number times the factorial of the number minus one.

Writing a recursive function in Scala is much like writing any function, except the return-type inference takes a vacation—Scala insists on seeing the return-type explicitly for recursive functions. The reason for this is, since the function calls itself in at least one path of execution, Scala doesn't want to take the burden of figuring out the return type.

Let's run the factorial() function for a relatively small input value:

```
println(factorial(5))
```

The call will run quickly and produce the desired result, showing that Scala handled the recursive call quite well:

```
120
```

Take a close look at the code on line 5 in the factorial() function; the last operation is the multiplication (*). In each call through the function, the value of the number parameter will be held in a level of stack, while waiting for the result from the subsequent call to arrive. If the input parameter is 5, the call will get six levels deep before the recursion terminates.

The stack is a limited resource and can't grow boundlessly. For a large input value, simple recursion will run into trouble rather quickly. For example, try calling the factorial() function with a large value, like so:

```
println(factorial(10000))
```

Here's the fate of that call:

```
java.lang.StackOverflowError
```

It's quite sad that such a powerful and elegant concept meets such a terrible fate, incapable of taking on some real demand.

Most languages that support recursion have limitations on the use of recursion. Thankfully, some languages, like Scala, have some nice support to avoid these issues, as you'll see next.

Tail Call Optimization (TCO)

Although many languages support recursion, some compilers go a step further to optimize recursive calls. As a general rule, they convert recursions into iterations as a way to avoid the stack overflow issue.

Iterations don't face the stack overflow issue that recursions are prone to, but iterations aren't as expressive. With the optimization, we can write highly expressive and intuitive recursive code and let the compiler transform recursions into safer iterations before runtime—see *Structure and Interpretation of Computer Programs [AS96]* by Abelson and Sussman. Not all recursions, however, can be transformed into iterations. Only recursions that have a special structure—*tail recursions*—enjoy this privilege. Let's explore this further.

In the factorial() function, the final call on line 5 is multiplication. In a *tail* recursion, the final call will be to the function itself. In that case, the function call is said to be in the tail position. We'll rework the factorial() function to use tail recursion, but first let's explore the benefit of doing so using another example.

No Optimization for Regular Recursions

Scala performs optimizations for tail recursion but doesn't provide any optimization for regular recursions. Let's see the difference with an example.

In the next example, the mad() function throws an exception when it meets a parameter value of 0. Note that the last operation in the recursion is multiplication.

ProgrammingRecursions/mad.scala

```
def mad(parameter: Int) : Int = {
  if(parameter == 0)
    throw new RuntimeException("Error")
  else
    1 * mad(parameter - 1)
}

mad(5)
```

Here's an excerpt from the output of running the code:

```
java.lang.RuntimeException: Error
        at Main$$anon$1.mad(mad.scala:3)
        at Main$$anon$1.mad(mad.scala:5)
        at Main$$anon$1.mad(mad.scala:5)
        at Main$$anon$1.mad(mad.scala:5)
        at Main$$anon$1.mad(mad.scala:5)
        at Main$$anon$1.mad(mad.scala:5)
        at Main$$anon$1.<init>(mad.scala:8)
```

The exception stack trace shows that we're six levels deep in the call to the mad() function before it blew up. This is regular recursion at work, just the way we'd expect.

TCO to the Rescue

Not all languages that support recursion support TCO; for example, Java doesn't have support for TCO and all recursion, at the tail position or not, will face the same fate of stack overflow for large input. Scala readily supports TCO.

Let's change the mad() function to remove the redundant product of 1. That'd make the call tail recursive—the call to the function is the last, or in the tail position.

```
def mad(parameter: Int) : Int = {
  if(parameter == 0)
    throw new RuntimeException("Error")
  else
    mad(parameter - 1)
}

mad(5)
```

Let's look at the output from this modified version:

```
java.lang.RuntimeException: Error
        at Main$$anon$1.mad(mad2.scala:3)
        at Main$$anon$1.<init>(mad2.scala:8)
```

The number of recursive calls to the mad() function is the same in both the versions. However, the stack trace for the modified version shows we're only one level deep, instead of six levels, when the exception was thrown. That's due to the handy work of Scala's tail call optimization.

You can see this optimization firsthand by running the scala command with the -save option, like so: scala -save mad.scala. This will save the bytecode in a file named mad.jar. Then run jar xf mad.jar followed by javap -c -private Main\$\$anon\$1.class. This will reveal the bytecode generated by the Scala compiler.

Let's first look at the bytecode for the mad() function written as a regular recursion:

```
  private int mad(int);
    Code:
       0: iload_1
       1: iconst_0
       2: if_icmpne     15
       5: new           #14                 // class
java/lang/RuntimeException
       8: dup
       9: ldc           #16                 // String Error
      11: invokespecial #20                 // Method
```

```
java/lang/RuntimeException."<init>":(Ljava/lang/String;)V
      14: athrow
      15: iconst_1
      16: aload_0
      17: iload_1
      18: iconst_1
      19: isub
      20: invokespecial #22                 // Method mad:(I)I
      23: imul
      24: ireturn
```

Toward the end of the mad() method, the invokeSpecial bytecode, marked as line 20, shows the call is recursive. Now, modify the code to make it tail recursive and then take a peek at the generated bytecode using the earlier steps.

```
  private int mad(int);
    Code:
       0: iload_1
       1: iconst_0
       2: if_icmpne     15
       5: new           #14                 // class
java/lang/RuntimeException
       8: dup
       9: ldc           #16                 // String Error
      11: invokespecial #20                 // Method
java/lang/RuntimeException."<init>":(Ljava/lang/String;)V
      14: athrow
      15: iload_1
      16: iconst_1
      17: isub
      18: istore_1
      19: goto          0
```

Instead of the invokeSpecial we see a goto, which is a simple jump, indicating a simple iteration instead of a recursive method call—that's smart optimization without much effort on our part.

Ensuring TCO

The compiler automatically transformed the tail recursion into an iteration. This quiet optimization is nice, but a bit unsettling—there's no immediate visible feedback to tell. To infer, we'd have to either examine the bytecode or check if the code fails for large inputs. Neither of those is appealing.

Thankfully, Scala has a nice little annotation to help program tail recursions with intention. You can mark any function with the annotation tailrec and Scala will verify at compile time that the function is tail recursive. If by mistake

the function is not tail recursive, and hence can't be optimized, the compiler will give a stern error.

To see this annotation at work, place it on the factorial() function, like so:

```
@scala.annotation.tailrec
def factorial(number: Int) : BigInt = {
  if(number == 0)
    1
  else
    number * factorial(number - 1)
}

println(factorial(10000))
```

Since this version of the factorial() function is a regular recursion and not a tail recursion, the compiler will report an appropriate error:

```
error: could not optimize @tailrec annotated method factorial: it contains
a recursive call not in tail position
    number * factorial(number - 1)
             ^
error found
```

Turning the regular recursion into tail recursion is not hard. Instead of performing the multiplication operation upon return from the recursive call to the method, we can pre-compute it and push the partial result as a parameter. Let's refactor the code for that:

```
@scala.annotation.tailrec
def factorial(fact: BigInt, number: Int) : BigInt = {
  if(number == 0)
    fact
  else
    factorial(fact * number, number - 1)
}

println(factorial(1, 10000))
```

The factorial() function in this modified version takes two parameters, with fact —the partial result computed so far—as the first parameter. The recursive call to the factorial() function is in the tail position. which complies with the annotation at the top of the function. With this change, Scala won't complain; instead it will apply the optimization for the call.

Run this version of the function to see the output:

```
2846259680917054518906413212119868890148051401702799230794179994274411340000
...
```

The TCO in Scala kicks in automatically for any tail recursive functions. The annotation is optional and, when used, makes the intention clear and expressive. It is a good idea to use the annotation. It ensures that the tail recursion stays that way through refactorings and also conveys the intent to fellow programmers who may come to refactor the code at a later time.