Tail-Recursive Algorithms in Scala

“Tail recursion is its own reward.”

From the “Functional” cartoon on xkcd.com.

Goals

The main goal of this lesson is to solve the problem shown in the previous lessons: Simple recursion creates a series of stack frames, and for algorithms that require deep levels of recursion, this creates a

`StackOverflowError` (and crashes your program).

"Tail recursion" to the rescue

Although the previous lesson showed that algorithms with deep levels of recursion can crash with a `StackOverflowError`, all is not lost. With Scala you can work around this problem by making sure that your recursive functions are written in a *tail-recursive* style.

A tail-recursive function is just a function whose *very last action* is a call to itself. When you write your recursive function in this way, the Scala compiler can optimize the resulting JVM bytecode so that the function requires only one stack frame — as opposed to one stack frame for each level of recursion!

On Stack Overflow, Martin Odersky explains tail-recursion in Scala:

> *"Functions which call themselves as their last action are called tail-recursive. The Scala compiler detects tail recursion and replaces it with a jump back to the beginning of the function, after updating the function parameters with the new values ... as long as the last thing you do is calling yourself, it's automatically tail-recursive (i.e., optimized)."*

But that `sum` function looks tail-recursive to me ...

"Hmm," you might say, "if I understand Mr. Odersky's quote, the `sum` function you wrote at the end of the last lesson sure looks tail-recursive to me":

```
def sum(list: List[Int]): Int = list match {
    case Nil => 0
    case x :: xs => x + sum(xs)
}
                        - - - -
                                  last action?
```

"Isn't the 'last action' a call to itself, making it tail-recursive?"

If that's what you're thinking, fear not, that's an easy mistake to make. But the answer is no, this function is not tail-recursive. Although `sum(tail)` is at the end of the second `case` expression, you have to think like a compiler here, and when you do that you'll see that the last two actions of this function are:

1. Call `sum(xs)`
2. When that function call returns, add its value to `x` and return that result

When I make that code more explicit and write it as a series of one-line statements, you see that it looks like this:

```
val s = sum(xs)
val result = x + s
return result
```

As shown, the last calculation that happens before the `return` statement is that the sum of `x` and `s` is calculated. If you're not 100% sure that you believe that, there are a few ways you can prove it to yourself.

1) Proving it with the previous "stack trace" example

One way to "prove" that the `sum` algorithm is not tail-recursive is with the "stack trace" output from the previous lesson. The JVM output shows the `sum` method is called once for each step in the recursion, so it's clear that the JVM feels the need to create a new instance of `sum` for each element in the collection.

2) Proving it with the @tailrec annotation

A second way to prove that `sum` isn't tail-recursive is to attempt to tag the function with a Scala annotation named `@tailrec`. This annotation won't compile unless the function is tail-recursive. (More on this later in this lesson.)

If you attempt to add the `@tailrec` annotation to `sum`, like this:

```
// need to import tailrec before using it
import scala.annotation.tailrec

@tailrec
def sum(list: List[Int]): Int = list match {
    case Nil => 0
    case x :: xs => x + sum(xs)
}
```

the `scalac` compiler (or your IDE) will show an error message like this:

```
Sum.scala:10: error: could not optimize @tailrec annotated method sum:
it contains a recursive call not in tail position
def sum(list: List[Int]): Int = list match {
                                ^
```

This is another way to "prove" that the Scala compiler doesn't think `sum` is tail-recursive.

Note: The text, "it contains a recursive call not in tail position," is the Scala error message you'll see whenever a function tagged with `@tailrec` isn't really tail-recursive.

So, how do I write a tail-recursive function?

Now that you know the current approach isn't tail-recursive, the question becomes, "How do I make it tail-recursive?"

A common pattern used to make a recursive function that "accumulates a result" into a tail-recursive function is to follow a series of simple steps:

1. Keep the original function signature the same (i.e., `sum`'s signature).
2. Create a second function by (a) copying the original function, (b) giving it a new name, (c) making it `private`, (d) giving it a new "accumulator" input parameter, and (e) adding the `@tailrec` annotation to it.
3. Modify the second function's algorithm so it uses the new accumulator. (More on this shortly.)
4. Call the second function from inside the first function. When you do this you give the second function's accumulator parameter a "seed" value (a little like the *identity* value I wrote about in the previous lessons).

Let's jump into an example to see how this works.

Example: How to make `sum` tail-recursive

1) Leave the original function signature the same

To begin the process of converting the recursive `sum` function into a *tail-recursive* `sum` algorithm, leave the external signature of `sum` the same as it was before:

```
def sum(list: List[Int]): Int = ...
```

2) Create a second function

Now create the second function by copying the first function, giving it a new name, marking it `private`, giving it a new "accumulator" parameter, and adding the `@tailrec` annotation to it. The highlights in this image show the changes:

```
// note: this code won't compile yet
@tailrec
private def sumWithAccumulator(list: List[Int], accumulator: Int): Int = list match {
    case Nil => 0
    case x :: xs => x + sum(xs)
}
```

This code won't compile as shown, so I'll fix that next.

> *Before moving on, notice that the data type for the accumulator (`Int`) is the same as the data type held in the `List` that we're iterating over.*

3) Modify the second function's algorithm

The third step is to modify the algorithm of the newly-created function to use the accumulator parameter. The easiest way to explain this is to show the code for the solution, and then explain the changes. Here's the source code:

```
@tailrec
private def sumWithAccumulator(list: List[Int], accumulator: Int): Int = {
    list match {
        case Nil => accumulator
        case x :: xs => sumWithAccumulator(xs, accumulator + x)
    }
}
```

Here's a description of how that code works:

- I marked it with `@tailrec` so the compiler can help me by verifying that my code truly is tail-recursive.
- `sumWithAccumulator` takes two parameters, `list: List[Int]`, and `accumulator: Int`.
- The first parameter is the same list that the `sum` function receives.
- The second parameter is new. It's the "accumulator" that I mentioned earlier.
- The inside of the `sumWithAccumulator` function looks similar. It uses the same match/case approach that the original `sum` method used.
- Rather than returning `0`, the first `case` statement returns the `accumulator` value when the `Nil` pattern is matched. (More on this shortly.)

- The second `case` expression is tail-recursive. When this case is matched it immediately calls `sumWithAccumulator`, passing in the `xs` (tail) portion of `list`. What's different here is that the second parameter is the sum of the `accumulator` and the head of the current list, `x`.
- Where the original `sum` method passed itself the tail of `xs` and then later added that result to `x`, this new approach keeps track of the accumulator (total sum) value as each recursive call is made.

The result of this approach is that the "last action" of the `sumWithAccumulator` function is this call:

```
sumWithAccumulator(xs, accumulator + x)
```

Because this last action really is a call back to the same function, the JVM can optimize this code as Mr. Odersky described earlier.

4) Call the second function from the first function

The fourth step in the process is to modify the original function to call the new function. Here's the source code for the new version of `sum`:

```
def sum(list: List[Int]): Int = sumWithAccumulator(list, 0)
```

Here's a description of how it works:

- The `sum` function signature is the same as before. It accepts a `List[Int]` and returns an `Int` value.
- The body of `sum` is just a call to the `sumWithAccumulator` function. It passes the original `list` to that function, and also gives its `accumulator` parameter an initial seed value of `0`.

Note that this "seed" value is the same as the *identity* value I wrote about in the previous recursion lessons. In those lessons I noted:

- The identity value for a sum algorithm is 0.
- The identity value for a product algorithm is 1.
- The identity value for a string concatenation algorithm is `""`.

A few notes about `sum`

Looking at `sum` again:

```
def sum(list: List[Int]): Int = sumWithAccumulator(list, 0)
```

a few key points about it are:

- Other programmers will call `sum`. It's the "Public API" portion of the solution.
- It has the same function signature as the previous version of `sum`. The benefit of this is that other programmers won't have to provide the initial seed value. In fact, they won't know that the internal algorithm uses a seed value. All they'll see is `sum`'s signature:

  def sum(list: List[Int]): Int

A slightly better way to write `sum`

Tail-recursive algorithms that use accumulators are typically written in the manner shown, with one exception: Rather than mark the new accumulator function as `private`, most Scala/FP developers like to put that function *inside* the original function as a way to limit its scope.

> *When doing this, the thought process is, "Don't expose the scope of `sumWithAccumulator` unless you want other functions to call it."*

When you make this change, the final code looks like this:

```
// tail-recursive solution
def sum(list: List[Int]): Int = {
    @tailrec
    def sumWithAccumulator(list: List[Int], currentSum: Int): Int = {
        list match {
            case Nil => currentSum
            case x :: xs => sumWithAccumulator(xs, currentSum + x)
        }
    }
    sumWithAccumulator(list, 0)
}
```

Feel free to use either approach. (Don't tell anyone, but I prefer the first approach; I think it reads more easily.)

A note on variable names

If you don't like the name `accumulator` for the new parameter, it may help to see the function with a different name. For a "sum" algorithm a name like `runningTotal` or `currentSum` may be more meaningful:

```
// tail-recursive solution
def sum(list: List[Int]): Int = {
    @tailrec
    def sumWithAccumulator(list: List[Int], currentSum: Int): Int = {
        list match {
            case Nil => currentSum
            case x :: xs => sumWithAccumulator(xs, currentSum + x)
        }
    }
    sumWithAccumulator(list, 0)
}
```

I encourage you to use whatever name makes sense to you. Personally I prefer `currentSum` for this algorithm, but you'll often hear this approach referred to as using an "accumulator," which is why I used that name first.

> *Of course you can also name the inner function whatever you'd like to call it.*

Proving that this is tail-recursive

Now let's prove that the compiler thinks this code is tail-recursive.

First proof

The first proof is already in the code. When you compile this code with the `@tailrec` annotation and the compiler doesn't complain, you know that the compiler believes the code is tail-recursive.

Second proof

If for some reason you don't believe the compiler, a second way to prove this is to add some debug code to the new `sum` function, just like we did in the previous lessons. Here's the source code for a full

Scala `App` that shows this approach:

```
import scala.annotation.tailrec

object SumTailRecursive extends App {

    // call sum
    println(sum(List.range(1, 10)))

    // the tail-recursive version of sum
    def sum(list: List[Int]): Int = {
        @tailrec
        def sumWithAccumulator(list: List[Int], currentSum: Int): Int = {
            list match {
               case Nil => {
                    val stackTraceAsArray = Thread.currentThread.getStackTrace
                    stackTraceAsArray.foreach(println)
                    currentSum
                }
                case x :: xs => sumWithAccumulator(xs, currentSum + x)
            }
        }
        sumWithAccumulator(list, 0)
    }

}
```

> *Note: You can find this code at this Github link. This code includes a few ScalaTest tests, including one test with a `List` of 100,000 integers.*

When I compile that code with `scalac`:

```
$ scalac SumTailRecursive.scala
```

and then run it like this:

```
$ scala SumTailRecursive
```

I get a lot of output, but if I narrow that output down to just the `sum`-related code, I see this:

```
[info] Running recursion.TailRecursiveSum
java.lang.Thread.getStackTrace(Thread.java:1552)
```

```
recursion.TailRecursiveSum$.sumWithAccumulator$1(TailRecursiveSum.scala:16)
recursion.TailRecursiveSum$.sum(TailRecursiveSum.scala:23)
//
// lots of other stuff here ...
//
scala.App$class.main(App.scala:76)
recursion.TailRecursiveSum$.main(TailRecursiveSum.scala:5)
recursion.TailRecursiveSum.main(TailRecursiveSum.scala)
45
```

As you can see, although the `List` in the code has 10 elements, there's only one call to `sum`, and more importantly in this case, only one call to `sumAccumulator`. You can now safely call `sum` with a list that has 10,000 elements, 100,000 elements, etc., and it will work just fine without blowing the stack. (Go ahead and test it!)

> *Note: The upper limit of a Scala `Int` is `2,147,483,647`, so at some point you'll create a number that's too large for that. Fortunately a `Long` goes to `2^63-1` (which is `9,223,372,036,854,775,807`), so that problem is easily remedied. (If that's not big enough, use a `BigInt`.)*

Summary

In this lesson, I:

- Defined tail recursion
- Introduced the `@tailrec` annotation
- Showed how to write a tail-recursive function
- Showed a formula you can use to convert a simple recursive function to a tail-recursive function

What's next

This lesson covered the basics of converting a simple recursive function into a tail-recursive function. I'm usually not smart enough to write a tail-recursive function right away, so I usually write my algorithms using simple recursion, then convert them to use tail-recursion.

To help in your efforts, the next lesson will show more examples of tail-recursive for different types of algorithms.

See also

- My list of Scala recursion examples
- Martin Odersky explaining tail recursion on Stack Overflow

< A Visual Look at JVM Stacks and Frames

Introduction to ScalaCheck, Part 1 >

Links:

This website uses cookies: learn more

alvinalexander.com is owned and operated by Valley Programming, LLC

In regards to links to Amazon.com, “As an Amazon Associate
I (Valley Programming) earn from qualifying purchases”