Extracted from:

Functional Programming Patterns in Scala and Clojure

Write Lean Programs for the JVM

The Pragmatic Bookshelf

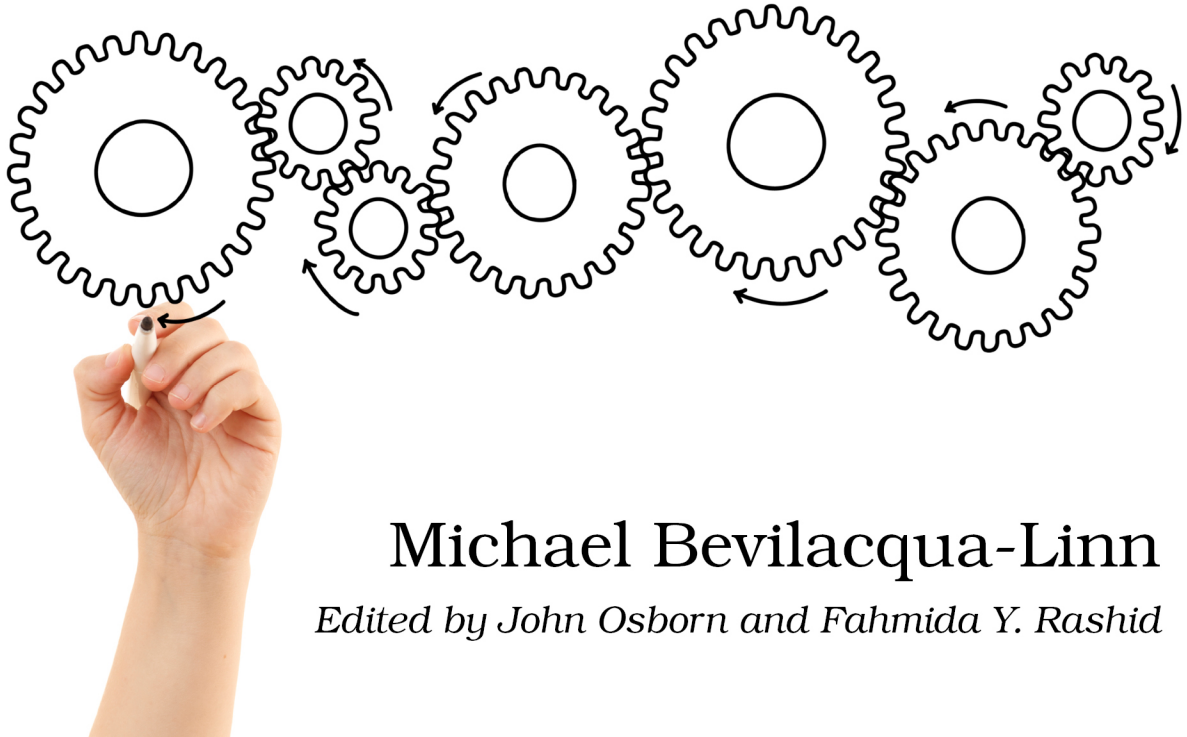
Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Functional Programming Patterns

in Scala and Clojure

Write Lean Programs for the JVM



Michael Bevilacqua-Linn

Edited by John Osborn and Fahmida Y. Rashid

Functional Programming Patterns in Scala and Clojure

Write Lean Programs for the JVM

Michael Bevilacqua-Linn

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

Pattern 12

Tail Recursion

Intent

To repeat a computation without using mutable state and without overflowing the stack

Overview

Iteration is an imperative technique that requires mutable state. For example, let's examine a trivial problem, writing a function that will calculate the sum from one up to an arbitrary number, inclusive. The code below does just that, but it requires both `i` and `sum` to be mutable:

JavaExamples/src/main/java/com/mblinn/functional/tailrecursion/Sum.java

```
public static int sum(int upTo) {
        int sum = 0;
        for (int i = 0; i <= upTo; i++)
                sum += i;
        return sum;
}
```

Since the functional world emphasizes immutability, iteration is out. In its place, we can use recursion, which does not require immutability. Recursion has its own problems, though; in particular, each recursive call will lead to another frame on the program's call stack.

To get around that, we can use a particular form of recursion called *tail recursion*, which can be optimized to use a single frame on the stack, a process known as *tail call optimization* or *TCO*.

Let's think about how we'd write the `sum()` as a recursive function, `sumRecursive()`. First, we need to decide when our recursion should stop and start. Since we're summing together all numbers stopping at some arbitrary number, it makes sense to work down from that number and stop at zero. This stopping point is known as our *base case*.

Next we need to figure out what to do to perform the actual computation. In this case, we take the number we're currently working on and add it to the results of calling `tailRecursive()` with that number minus one. Eventually, we get down to our base case of zero, at which point the stack unwinds, returning

partial sums as it goes, until it reaches the top and returns the final sum. The code below demonstrates this solution:

JavaExamples/src/main/java/com/mblinn/functional/tailrecursion/Sum.java

```
public static int sumRecursive(int upTo) {
        if (upTo == 0)
                return 0;
        else
                return upTo + sumRecursive(upTo - 1);
}
```

There's a problem with this, though. Each recursive call adds a frame to the stack, which means this solution takes memory proportional to the size of the sequence we're summing, as shown in the following figure.

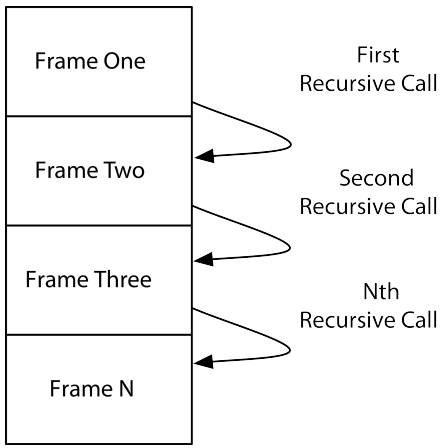


Figure 10—Simple Stack. An illustration of the stack during normal recursive calls—each recursive call adds a call to the stack; these frames represent memory that cannot be reclaimed until after the recursion is done.

Clearly this isn't practical, but we can do better. The ultimate cause for exploding stack use is that each time we make a recursive call, we need the result of that call to finish the computation we're doing in the current call. This means that the runtime has no choice but to store the intermediate results on the stack.

If we were to make sure that the recursive call was the last thing that happens in each branch of the function, known as the tail position, this would no longer be the case. Doing so requires us to take the intermediate values that were formerly stored on the stack and pass them through the call chain. The code below illustrates this:

JavaExamples/src/main/java/com/mblinn/functional/tailrecursion/Sum.java

```
public static int sumTailRecursive(int upTo, int currentSum) {
        if (upTo == 0)
                return currentSum;
        else
                return sumTailRecursive(upTo - 1, currentSum + upTo);
}
```

Once we rewrite the function to be tail recursive, it's possible to use TCO to run it in only a single stack frame, as shown in this figure.

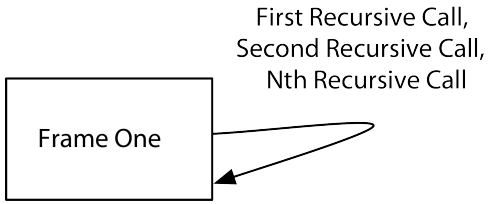


Figure 11—Stack with TCO. With TCO, recursive calls in the tail position don't generate a new stack frame. Instead, each call uses the existing stack frame, removing whatever data was there from the previous call.

Unfortunately, the JVM doesn't support TCO directly, so Scala and Clojure need to use some tricks to compile their tail recursive calls down to the same bytecode used for iteration. In Clojure's case, this is done by providing two special forms, `loop` and `recur`, instead of using general purpose function calls.

In Scala's case, the Scala compiler will attempt to translate tail recursive calls into iteration behind the scenes, and Scala provides an annotation, `@tailrec`, that can be placed on functions that are meant to be used in a tail recursive manner. If the function is called recursively without being in the tail position, the compiler will generate an error.

Code Sample: Recursive People

Let's take a look at a recursive solution to a simple problem. We've got a sequence of first names and a sequence of last names, and we want to put them together to make people. To solve this, we need to go through both sequences in lock step. We'll assume that some other part of the program has verified that the two sequences are of the same size.

At each step in the recursion, we'll take the first element in both sequences and put them together to form a full name. We'll then pass the rest of each sequence onto the next recursive call along with a sequence of the people we've formed so far. Let's see how it looks in Scala.

In Scala

The first thing we'll need is a Scala case class to represent our people. Here we've got one with a first and a last name:

ScalaExamples/src/main/scala/com/mblinn/mbfpp/functional/tr/Names.scala
```
case class Person(firstNames: String, lastNames: String)
```

Next up is our recursive function itself. This is actually split into two functions. The first is a function named makePeople, which takes in two sequences, firstNames and lastNames. The second is a helper function nested inside of makePeople, which adds an additional argument used to pass the list of people through recursive calls. Let's take a look at the whole function before we break it down into smaller parts:

ScalaExamples/src/main/scala/com/mblinn/mbfpp/functional/tr/Names.scala
```
def makePeople(firstNames: Seq[String], lastNames: Seq[String]) = {
  @tailrec
  def helper(firstNames: Seq[String], lastNames: Seq[String],
                      people: Vector[Person]): Seq[Person] =
    if (firstNames.isEmpty)
      people
    else {
      val newPerson = Person(firstNames.head, lastNames.head)
      helper(firstNames.tail, lastNames.tail, people :+ newPerson)
    }
  helper(firstNames, lastNames, Vector[Person]())
}
```

First, let's examine the function signature of makePeople:

```
def makePeople(firstNames: Seq[String], lastNames: Seq[String]) = {
        «function-body»
}
```

This just says that makePeople takes two Seqs of String. Since we don't specify a return type, the compiler will infer it from the function body.

Next up, let's look at the signature of the helper function. This function is responsible for the actual tail recursive calls. The helper function is annotated with a @tailrec annotation, which makes the compiler generate an error if it's called recursively but not tail recursively. The function signature simply adds an additional argument, the people vector, which will accumulate results through recursive calls.

Notice that we specified a return type here, though we generally omit it in our Scala examples. This is because the compiler can't infer types for recursively called functions.

```
def helper(firstNames: Seq[String], lastNames: Seq[String],
                        people: Vector[Person]): Seq[Person] =
                        «function-body»
```

Now the body for helper. If the firstNames sequence is empty, we return the list of people we've built up. Otherwise, we pick the first first name and the first last name off of their respective sequences, create a Person out of them, and call the helper function again with the tail of the two sequences and the new person appended to the list of people:

```
if (firstNames.isEmpty)
        people
else {
        val newPerson = Person(firstNames.head, lastNames.head)
        helper(firstNames.tail, lastNames.tail, people :+ newPerson)
}
```

Finally, we simply call helper with the sequences of names and an empty Vector to hold our people:

```
helper(firstNames, lastNames, Vector[Person]())
```

One closing note on the syntax: using some of Scala's object-oriented features, namely methods, would let us cut out some of the verbosity that comes along with a recursive function definition. The method signatures would look like this:

```
def makePeopleMethod(firstNames: Seq[String], lastNames: Seq[String]) = {
        @tailrec
        def helper(firstNames: Seq[String], lastNames: Seq[String],
                        people: Vector[Person]): Seq[Person] =
                        «method-body»
        }
}
```

Since we're sticking mainly to the functional bits of Scala in this book, we're using functions for most of the examples rather than methods. Methods can often be used as higher-order functions in Scala, but it can sometimes be awkward to do so.

In Clojure

In Clojure, tail recursive calls are never optimized, so even a tail recursive call will end up consuming a stack frame. Instead of providing TCO, Clojure gives us two forms, loop and recur. The loop form defines a recursion point, and the keyword recur jumps back to it, passing it new values.

In practice, this looks almost exactly like defining a private helper function does, so the form of our solution is very similar to the Scala solution, though

we'll use a simple map to store our people, as is standard in Clojure. Let's take a look at the code:

ClojureExamples/src/mbfpp/functional/tr/names.clj
```
(defn make-people [first-names last-names]
  (loop [first-names first-names last-names last-names people []]
    (if (seq first-names)
      (recur
        (rest first-names)
        (rest last-names)
        (conj
          people
          {:first (first first-names) :last (first last-names)}))
      people)))
```

The first interesting bit of code here is the loop declaration. Here, we define our recursion point and the values we'll start our recursion at: the passed-in sequences of first and last names and an empty vector we'll use to accumulate people as we recur.

```
(loop [first-names first-names last-names last-names people []]
        «loop-body»
)
```

The code snippet first-names first-names last-names last-names people [] might look a little funny, but all it's doing is initializing the first-names and last-names that we're defining in the loop to be the values that were passed into the function and the people to an empty vector.

The bulk of the example is in the if expression. If the sequence of first names still has items in it, then we take the first item from each sequence, create a map to represent the person, and conj it onto our people accumulator.

Once we've conjed the new person onto the people vector, we use recur to jump back to the recursion point we defined with loop. This is analogous to the recursive call that we made in the Scala example.

If we don't jump back, we know that we've gone through the sequences of names, and we return the people we've constructed.

```
(if (seq first-names)
        (recur
                (rest first-names)
                (rest last-names)
                (conj
                        people
                        {:first (first first-names) :last (first last-names)}))
        people)
```

It may not be immediately apparent why the test in the if expression above works. It's because the seq of an empty collection is nil, which evaluates to false, while the seq of any other collection yields a nonempty sequence. The snippet below demonstrates this:

```
=> (seq [])
nil
=> (seq [:hi])
(:hi)
```

Using nil as the base case for a recursion when you're dealing with sequences is common in Clojure.

Discussion

Tail recursion is equivalent to iteration. In fact, the Scala and Clojure compilers will compile their respective ways of handling tail recursion down to the same sort of bytecode that iteration in Java would. The main advantage of tail recursion over iteration is simply that it eliminates a source of mutability in the language, which is why it's so popular in the functional world.

I personally prefer tail recursion over iteration for a couple of other minor reasons. The first is that it eliminates an extra index variable. The second is that it makes it explicit exactly what data structures are being operated on and what data structures are being generated, because they're both passed as arguments through the call chain.

In an iterative solution, if we were trying to operate on two sequences in lock step and generate another data structure, they would all just be mixed in with the body of a function that may be doing other things. I've found that using tail recursion over iteration acts as a nice forcing factor to structure our functions well, since all of the data we're operating on must be passed through the call chain, and it's hard to do that if you've got more than a few pieces of data.

Since tail recursion is equivalent to iteration, it's really a fairly low-level operation. There's generally some higher-level, more-declarative way to solve a problem than using tail recursion. For instance, here's a shorter version of the solution to our person-making example that takes advantage of some higher-order functions in Clojure:

ClojureExamples/src/mbfpp/functional/tr/names.clj

```
(defn shorter-make-people [first-names last-names]
  (for [[first last] (partition 2 (interleave first-names last-names))]
    {:first first :last last}))
```

Which solution to use is a matter of preference, but experienced functional programmers tend to prefer the shorter, more-declarative solutions. They're easier for the experienced functional programmer to read at a glance. The downside to these solutions is that they're harder for the novice to grok, since they may require knowledge of many higher-order library functions.

Whenever I'm about to write a solution that requires tail recursion, I like to comb the API docs for higher-order functions, or a combination of higher-order functions, that do what I want. If I can't find a higher-order function that works, or if the solution I come up with involves many higher-order functions combined in Byzantine ways, then I fall back to tail recursion.

Related Patterns

Pattern 5, *Replacing Iterator*, on page ?

Pattern 13, *Mutual Recursion*, on page ?

Pattern 14, *Filter-Map-Reduce*, on page ?