Matt Malone’s Old-Fashioned Software Development Blog

September 27, 2008

Tail-Recursion Basics In Scala

Posted by Matt under Scala | Tags: scala recursion |
[15] Comments

Recursion 101

We all know what recursion is, right? A function calls itself. Or function A calls function B which calls function A. Or A calls B, which calls C, which calls A, etc. By far the most common situation is that in which a function calls itself.

You don’t see a lot of recursive code in Java. There are a couple reasons for this. First, recursion is hard. It’s not intuitive. With iteration (the main alternative to recursion) you see the big picture. You can look at the whole loop and its easy to understand even for the beginner. With recursion, you see one layer and you have to imagine what happens when those layers stack up. Like anything else, if you practice iteration then it becomes easier, but compared to iteration, recursion is hard to learn.

Second, Java is not designed to accomodate recursion. It’s designed to accomodate iteration. Java gives you for loops, for-each loops, while loops, do loops, arrays, Iterators, ResultSets, etc. These constructs are all about iteration. Plus, recursion in Java has an Achilles’ heel: the call stack.

Generally, when you call a function in any language a new level is added to the call stack. The call stack, we all know, is what keeps track of local variables, what function has called what, etc. It’s no different in Java. But the stack has a finite and limited size. Recursion is fine if you know for certain that you’ll never go more than a few dozen levels deep. But if the recursion goes too deep, you run out of stack space and your program goes kaput. That doesn’t happen with iteration, so it’s safer to just avoid recursion altogether.

Recursion In Scala

Scala, however, being a functional language is *very much* geared toward recursion rather than iteration. So how does it overcome the limitations of the call stack? Let's look at an example recursive function in scala.

```
def listLength1(list: List[_]): Int = {
  if (list == Nil) 0
  else 1 + listLength1(list.tail)
}

var list1 = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
var list2 = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
1 to 15 foreach( x => list2 = list2 ++ list2 )

println( listLength1( list1 ) )
println( listLength1( list2 ) )
```

Function listLength1 recursively counts the number of items in a list. Try running this in the interpreter. It works fine for list1, the short list, but the longer list exhausts the stack. Recursion is a functional language's bread and butter, but we see here that even scala, a functional language, is subject to call stack limitations.

Don't give up on recursion yet, though. Scala has a very important optimization that will allow you to recurse without limit provided you use the right kind of recursion.

Head Recursion And Tail Recursion

There are two basic kinds of recursion: head recursion and tail recursion. In head recursion, a function makes its recursive call and then performs some more calculations, maybe using the result of the recursive call, for example. In a tail recursive function, all calculations happen first and the recursive call is the last thing that happens.

The importance of this distinction doesn't jump out at you, but it's extremely important! Imagine a tail recursive function. It runs. It completes all its computation. As its very last action, it is ready to make its recursive call. What, at this point, is the use of the stack frame? None at all. We don't need our local variables anymore because we're done with all computations. We don't need to know which function we're in because we're just going to re-enter the very same function. Scala, in the case of tail recursion, can eliminate the creation of a new stack frame and just re-use the current stack frame. The stack never gets any deeper, no matter how many times the recursive call is made. That's the voodoo that makes tail recursion special in scala.

Incidentally, some languages achieve a similar end by converting tail recursion into iteration rather than by manipulating the stack.

This won't work with head recursion. Do you see why? Imagine a head recursive function. First it does some work, then it makes its recursive call, then it does a little more work. We can't just re-use the current stack frame when we make that recursive call. We're going to NEED that stack frame info after the recursive call completes. It has our local variables, including the result (if any) returned by the recursive call.

Here's a question for you. Is the example function listLength1 head recursive or tail recursive? Well, what does it do? (A) It checks whether its parameter is Nil. (B) If so, it returns 0 since Nil has 0 length. (C) If not, it returns 1 plus the result of a recursive call. The recursive call is the last thing we typed before ending the function. That's tail recursion, right? Wrong. The recursive call is made, and THEN 1 is added to the result, and this sum is returned. This is actually head recursion (or middle recursion, if you like) because the recursive call is not the very last thing that happens.

Tail Recursion Example

When you write a recursive function in scala, your aim is to encourage the compiler to make tail recursion optimizations. Now let's rewrite that function using tail recursion.

```
def listLength2(list: List[_]): Int = {
  def listLength2Helper(list: List[_], len: Int): Int = {
    if (list == Nil) len
    else listLength2Helper(list.tail, len + 1)
  }
  listLength2Helper(list, 0)
}

println( listLength2( list1 ) )
println( listLength2( list2 ) )
```

I wrote this as two functions (listLength2 and an internal helper function) to preserve the one-parameter interface used in the earlier example. It would be great if we could specify a default value for a parameter. We could then write this as one function, but I don't know a way to do it. Long story short: listLength2 just calls listLength2Helper which does the real work and is the recursive function.

Is listLength2Helper head recursive or tail recursive? When the recursive call is made are we *really* finished with the stack frame, allowing scala to optimize for tail recursion? Just like in listLength1, this function first checks for a Nil list, but this time returns the len parameter rather than 0. If list is non-Nil then we make the recursive call. But there's still an addition going on here, len + 1. Does that ruin the tail recursion? No. That term is evaluated first. Only after all the parameters have been evaluated does the recursive call happen. This function does indeed qualify for tail recursion optimization.

By The Way…

The two examples in this post are very elementary and it's easy to predict visually that tail recursion optimization will be applied. But it's not always so simple. If you're trying to write a complex recursive function and you require that it be optimized as tail recursive then you have a problem. How do you verify that you have succeeded? The only ways that I know of are (A) examine the resulting object code or (B) write a unit test to verify that the optimization was made. The problem with this is that you open yourself up to bugs in your unit tests. You have to be able to write a test that you *know* will fail if the function is not optimized. I'm new to scala, so maybe there's a better way to do this. If there is, please educate me.

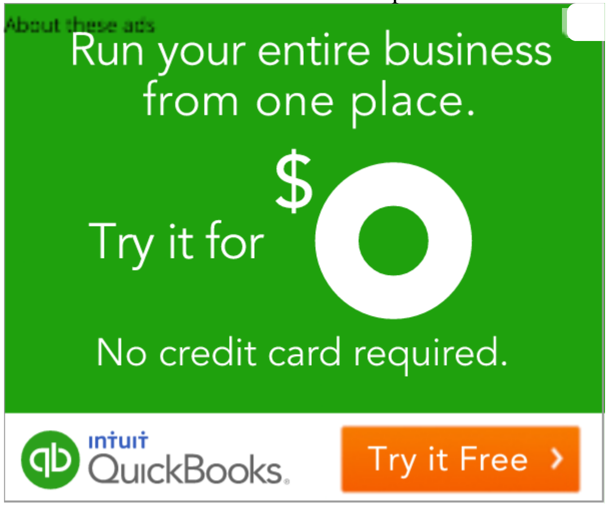
What I would like to see is an annotation that marks a function as *requiring* tail recursion optimization. Such an annotation would trigger a compiler error if the compiler were unable to make the optimization. The annotation I propose would work like this:

```scala
@TailRecursive
def listLength1(list: List[_]): Int = {
  if (list == Nil) 0
  else 1 + listLength1(list.tail)
}

def listLength2(list: List[_]): Int = {
  @TailRecursive
  def listLength2Helper(list: List[_], len: Int): Int = {
    if (list == Nil) len
    else listLength2Helper(list.tail, len + 1)
  }
  listLength2Helper(list, 0)
}
```

The compiler would succeed for listLength2Helper, but would report an error when it failed to apply the requested tail recursion optimization. This doesn't let the developer off the hook on unit testing, nor does it relieve the developer from having to code carefully. What it does is provide early, infallible verification that a crucial feature was implemented. Why force the developer to examine the classfile or write a bug-vulnerable unit test when the compiler *knows* the answer and could just cough up the information at compile time?

Don't forget to subscribe to my RSS feed, or follow this blog on Twitter.



15 Responses to “Tail-Recursion Basics In Scala”

1. Piotr Says:

   March 19, 2009 at 8:43 am
   @tailrec in upcomming Scala 2.8
   does what you write about

   http://www.nabble.com/-scala–%40tailrec-and-%40switch-now-in-trunk-p22543940.html

   it is in latest dev builds.

   Reply
2. Matt Says:

   March 19, 2009 at 3:44 pm
   Thank, Piotr. That’s great news. I hadn’t heard of it before. I’m going to try it out.

   Reply
   1. Artur Gajowy Says:

      January 28, 2010 at 9:53 pm
      Also, named parameters with default values are coming to Scala 2.8. My dreams come true, and so do yours, I think ;]

      Reply
3. Daniel Sobral Says:

   August 26, 2009 at 2:20 am
   Note that the Scala compiler can’t optimize every tail recursion. It can do so on functions, but not on non-final methods (because they can be overrriden).

   Reply

4. Tom Says:

   November 24, 2010 at 3:43 pm
   When I first started learning programming, I actually found that the recursive solution was usually the easiest to find, then I'd move on to the iterative approach. Recursion only deals with a single step. You think what have I got, what do I need to do to get to the next step. This was much easier than trying to see the forest through the trees and write everything in a while loop, constantly updating variables, etc. It isn't surprising that the recursive solution usually requires many fewer lines of code.

   Reply
5. Joakim Andersson » Blog Archive » Learning Scala: Recursion Says:

   January 19, 2011 at 10:15 pm
   […] Tail-Recursion Basics In Scala […]

   Reply
6. Tail recursive « Miała być Java Says:

   January 25, 2011 at 9:40 pm
   […] recursive'. Poszukiwanie tego co to znaczy doprowadziło mnie między innymi do tego wpisu. Jest tu w miarę łopatologiczne wytłumaczenie o co chodzi w tym. Upraszczając: za tym pojęciem […]

   Reply
7. pmcs Says:

   September 22, 2012 at 3:09 am
   I don't know if this was a shortcoming of earlier versions or not, but current (>= 2.9.2) versions of Scala support default parameter values. Your tail-recursive function listLength2 can be rewritten without a helper function as follows:

   ```
   def listLength2(list: List[_], len: Int = 0): Int = {
   if (list == Nil) len
   else listLength2(list.tail, len + 1)
   }
   ```

   Reply
8. nicholassterling Says:

   November 18, 2012 at 12:38 am
   @pmcs: There are a couple of caveats with using default arguments in that way. First, users can actually supply the argument, purposely or accidentally, and then your code will do something it could never do before. You may or may not find that desirable. In any case the new parameter should be documented as part of the API.

   Secondly, methods defined within another method are private, so they cannot be overridden, making tail-recursion optimization possible. A public, non-final method cannot receive the optimization, so if you do what are suggesting and want the method to be tail-recursive it will have to be private and/or final. Of course doing either has implications for those using the class, so be careful.

   In general I would say that it is best not to do this.

Reply

9. Narada Says:

   September 17, 2013 at 7:43 pm

   Excellent article. Very well written and explained in very simple terms. It's a skill. Thanks.

   Reply

10. Jacob Lorensen Says:

    November 27, 2013 at 9:20 pm

    Caveat: I'm not familiar with the JVM internals, so this may be complete bullshit.

    But I really wonder what the *** final / non-final method has to do with tail call elimination? When you set up a call to a method, you allocate/create the stack frame and call the method.

    If the instruction executed immediately after return is itself a return, this call could instead be optimized into variable assignmails plus "jump" instruction, avoiding the stack frame creation. And that is regardless of how the called method's address is obtained (ie. if was known statically in advance or if it comes from a dynamic method lookup table).

    This really applies to the more general tail call elimination than just tail recursion call elimination.

    At least, that is my experience from small toy interpreters I've made for myself over the years.

    Reply

    1. Jacob Lorensen Says:

       November 27, 2013 at 9:27 pm

       Oh! I am silently assuming a C like calling convention, which may not apply to the JVM. Googling away…

       Reply

    2. Daniel Sobral Says:

       November 28, 2013 at 12:25 am

       This runs afoul of the byte code verifier of the virtual machine. Also, I don't think code layout must be preserved by the class loader, so you may not even be able to compute the address of any other method in advance, and even that presumes all the methods are in the same class file to begin with.

       Anyway, here's the restriction it would violate:

       The static constraints on the operands of instructions in the code array are as follows:

       The target of each jump and branch instruction (jsr, jsr_w, goto, goto_w, ifeq, ifne, ifle, iflt, ifge, ifgt, ifnull, ifnonnull, if_icmpeq, if_icmpne, if_icmple, if_icmplt, if_icmpge, if_icmpgt, if_acmpeq, if_acmpne) must be the opcode of an instruction within this method.

       See this link: http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.9

       Reply

3. nicholassterling Says:

   November 28, 2013 at 6:17 am
   The significance of private/final is that since this is a *compile-time* optimization by scalac rather than a jvm optimization, you can't know whether a method has been overridden in a subclass. If it has, then what looks like a method calling itself is really a call to the overriding method.

   http://stackoverflow.com/questions/4785502/why-wont-the-scala-compiler-apply-tail-call-optimization-unless-a-method-is-fin

   Of course the JVM itself *does* know whether a method has been overridden and could safely perform the optimization (and back it out later if a subclass is loaded which does such an override).

   Reply
   1. Jacob Lorensen Says:

      November 29, 2013 at 8:24 pm
      Thanks for explaining and giving links. Supporting this in the jvm seems like a coming requirement, given that so many functional-style programming languages gain followers. Not having proper TCO is… something of a showstopper when I met scala and clojure too, really. When I got to that section, my thought was "You're not seriously saying this language doesn't do TCO?". Now I know why.