Tail self-calls become loops

© Miguel Garcia, STS, Hamburg University of Technology

http://www.sts.tu-harburg.de/people/mi.garcia

December 31, 2009

Contents

1 Tail call optimization (TCO) in Scala 1
2 AST shapes that trigger the rewrite 2
  2.1 The DefDef case .................................................. 4
  2.2 Other cases .......................................................... 5
3 Interplay with inlining 5
4 Trampolines 7
5 Continuations 7

Abstract

Like in previous code walkthroughs, these notes provide details about the internal workings of the Scala compiler, this time covering how a specific form of recursion (tail self-calls) is transformed into looping, to keep stack space constant. The walkthrough allows understanding why some AST shapes don’t get optimized, as well as the interplay with inlining.

1 Tail call optimization (TCO) in Scala

The design aspects around TCO in Scala are common to all languages with strict evaluation of arguments. As of now the JVM does not support constant-space tail-calls (unlike Microsoft’s CLR) but this is about to change:

- http://www.ssw.uni-linz.ac.at/Research/Papers/Schwaighofer09Master/

The current version of scalac (2.8) rewrites a program as in Listing 1 into the one shown in Listing 2. The invocations to method loop() exhibit both:

- a tail self-recursive call, in the example “else loop(n - 1, acc * n)”, where tail means “last in its path in the control-flow graph”. This invocation is optimized into looping (in Listing 2, loop(...this,n,acc) is the label whose next instruction is the head of the loop).
Listing 1: Original program

```scala
package tailRecExperiment

object tailRexExperiment5 extends Application {
  def factorial(n: Int) = {
    def loop(n: Int, acc: Int): Int =
      if (n <= 0) acc
      else if (n == 1) 1 * loop(n - 1, acc * n)
      else loop(n - 1, acc * n)
    loop(n, 1)
  }
}
```

- a non-tail call (self-recursive in this case, but doesn’t matter) won’t be optimized: “else if (n == 1) 1 * loop(n - 1, acc * n)”. It’s non-tail because of the multiplication by one of the result of the invocation (constant elimination removes the multiplication by 1, as can be seen in ICODe after dce, but that’s too late: the recursive call was not TCO’d, Sec. 3).

2 AST shapes that trigger the rewrite

The implementation of method `override def transform(tree: Tree): Tree` in `TailCalls.scala` helps explain some features of TCO behavior in Scala. It applies the identity transformation to most tree shapes (by invoking `super.transform(tree)`).

In contrast, the following AST shapes are “custom-processed”:

```scala
case dd @ DefDef(mods, name, tparams, vparams, tpt, rhs) =>
case Block(stats, expr) =>
case CaseDef(pat, guard, body) =>
case If(cond, thenp, elsep) =>
case Match(selector, cases) =>
case Try(block, catches, finalizer) =>
  // no calls inside a try are in tail position,
  // but keep recursing for nested functions
  case Apply(tapply @ TypeApply(fun, targs), vargs) =>
  case Apply(fun, args) if (fun.symbol == definitions.Boolean_or ||
    fun.symbol == definitions.Boolean_and) =>
case Apply(fun, args) =>
```

For example, TCO is not applied inside the condition part in an if expression (the condition can’t contain a self call in tail position). The handler for If nodes:

```scala
case If(cond, thenp, elsep) =>
treeCopy.If(tree, cond, transform(thenp), transform(elsep))
```

attempts to apply TCO to both thenp and elsep, but cond is not transformed. Other AST shapes are rejected altogether:

```scala
case Alternative(_) | Star(_) | Bind(_, _) =>
  throw new RuntimeException("We should've never gotten inside a pattern")
```
Listing 2: After self-tail calls have been made into loops

```scala
package tailRecExperiment {

final class tailRecExperiment5 extends java.lang.Object with Application with ScalaObject {

  def this(): object tailRecExperiment.tailRecExperiment5 = {
    tailRecExperiment5.super.this();
  }

  def factorial(n: Int): Int = {
    def loop(n: Int, acc: Int): Int = {
      <synthetic> val _$this: tailRecExperiment.tailRecExperiment5.type = tailRecExperiment5.this;
      _loop(_$this,n,acc){
        if (n.<=(0))
          acc
        else
          if (n.==(1))
            1.*(loop(n.-(1), acc.*(n)))
          else
            _loop(tailRecExperiment5.this, n.-(1), acc.*(n))
      }
    }
    loop(n, 1)
  }
}
```

Finally, other AST shapes are not visited further but returned as-is:

```scala
case Super(qual, mix) => tree
  case This(qual) => tree
  case Select(qualifier, selector) => tree
  case Ident(name) => tree
  case Literal(value) => tree
  case TypeTree() => tree
  case _ => tree
```

The loop resulting from TCO application has as target a `LabelDef`, whose case class is depicted in Listing 3. `LabelDef`s can be added to the AST after invoking:

```scala
def LabelDef(sym: Symbol, params: List[Symbol], rhs: Tree): LabelDef =
  atPos(sym.pos) {
    LabelDef(sym.name, params map Ident, rhs) setSymbol sym
  }
```

3
/** Labelled expression - the symbols in the array (must be Idents!)
* are those the label takes as argument
* The symbol that is given to the labeldef should have a MethodType
* (as if it were a nested function)
* Jumps are apply nodes attributed with label symbol, the arguments
* will get assigned to the idents.
* Note: on 2005-06-09 Martin, Iuli, Burak agreed to have forward
* jumps within a Block.
*/
case class LabelDef(name: Name, params: List[Ident], rhs: Tree)
  extends DefTree with TermTree {
    assert(rhs.isTerm)
  }

2.1 The DefDef case

Things start to set in motion when visiting a DefDef. At this point, a (TailCalls-local) Context is created (to represent the method being visited) and its fields initialized from those in a previous context (the first context being a dummy, with fields set to NoSymbol). After the new context is there, it’s mutated. Just like so:

```scala
val newCtx = mkContext(ctx)
newCtx.currentMethod = tree.symbol
newCtx.makeLabel()
val currentClassParam =
  tree.symbol.newSyntheticValueParam(currentClass.typeOfThis)
newCtx.label.setInfo(
  MethodType(currentClassParam :: tree.symbol.tpe.params,
  tree.symbol.tpe.finalResultType))
newCtx.tailPos = true
```

At this stage, the method may be discarded for TCO purposes by setting isEligible to false, i.e. in case the method is neither final nor local to a module. For example, a method local to another method isEligible.

Afterwards, transform() is invoked recursively on the body of the method (i.e., on the rhs of the DefDef). The outcome of this attempt can be any of:

```scala
case newRHS if isEligible && newCtx.accessed => // pretty much the same DefDef is returned, but with a new body :-) case _ if recommend => // inform that the method *could have been* TCOed, and return original body case rhs => rhs // as-is
```

And that completes the case handler for DefDef. Still, the above does not explain how the method’s body was transformed, other than saying “transform()
is invoked recursively on the body of the method”. The new body itself is the 
Block(. . .) shown below, built within the case newRHS:

```plaintext
typed(atPos(tree.pos)(Block(
    List(ValDef(newThis, This(currentClass))),
    LabelDef(newCtx.label, newThis :: (vparams.flatten map (_.symbol)), newRHS))
))
```

The first argument, a length-one list of statements, declares and initializes a local variable newThis that will be an argument to the LabelDef that constitutes the expression in the block. What the enclosed invocation of transform() returned, newRHS, is the rhs of that LabelDef.

So “the other cases” listed at the beginning of Sec. 2 are responsible for building newRHS and updating newCtx (as necessary for evaluating the pattern guard case newRHS if isEligible && newCtx.accessed =>). BTW, newCtx is an alias for the ctx field of the current instance of TailCallElimination.

Those “other cases” are briefly covered next.

2.2 Other cases

When a statement is visited (which can happen to be a self-call), it’s necessary to know whether that statement is in a tail position. Before visiting children, that information is conveyed to the visitor, as shown below in the example of a Block (the “false” argument makes ctx.tailPos == false).

```plaintext
case Block(stats, expr) =>
    treeCopy.Block(tree,
    transformTrees(stats, mkContext(ctx, false)),
    transform(expr))
```

For other AST shapes, syntax alone determines which children can’t contain a self-call in tail-position. Besides the If example from the previous section, other examples in this category are:

```plaintext
case CaseDef(pat, guard, body) =>
    // no self-calls in tail-pos in the guard
    treeCopy.CaseDef(tree, pat, guard, transform(body))

case Apply(fun, args) if (fun.symbol == definitions.Boolean_or ||
    fun.symbol == definitions.Boolean_and) =>
    // don’t mess up with short-circuit evaluation
    treeCopy.Apply(tree, fun, transformTrees(args))
```

The two cases where a call is actually rewritten into a jump are shown in Listing 4 and in Listing 5.

That was brief, wasn’t it?

3 Interplay with inlining

Inlining takes place after the icode phase, and thus the TCO transformation cannot revisit decisions made there. As a result, the block of instructions being inlined (replacing a method call) can contain a recursive invocation.
Listing 4: Shape A that triggers TCO rewriting

```scala
case Apply(tapply @ TypeApply(fun, targs), vargs) =>
  lazy val defaultTree = treeCopy.Apply(tree, tapply, transformTrees(vargs, mkContext(ctx, false)));
  if (ctx.currentMethod.isFinal &&
    ctx.tailPos &&
    isSameTypes(ctx.tparams, targs map (_.tpe.typeSymbol)) &&
    isRecursiveCall(fun)) {
    fun match {
      case Select(receiver, _) =>
        val recTpe = receiver.tpe.widen
        val enclTpe = ctx.currentMethod.enclClass.typeOfThis
        // make sure the type of 'this' doesn't change through this polymorphic recursive call
        if (!forMSIL &&
            (receiver.tpe.typeParams.isEmpty ||
             (receiver.tpe.widen == ctx.currentMethod.enclClass.typeOfThis)))
          rewriteTailCall(fun, receiver :: transformTrees(vargs, mkContext(ctx, false)))
        else
          defaultTree
      case _ => rewriteTailCall(fun, This(currentClass) :: transformTrees(vargs, mkContext(ctx, false)))
    }
  } else
    defaultTree
```

The example in Listing 6, besides containing @inline annotations on methods declared final, was compiled with option -Yinline. Unlike other examples, in this case the Application object is not extended (in which case the code would execute as a static initialization). So all is set for inlining to do its job.

The invocation in the then branch of “if (n == 2)” will be replaced with another to loop() (which itself won’t be inlined). In this case, that branch will be visited only once during the whole execution of factorial(), but the case is illustrative. The debug view is shown in Figure 1.

To see the ICODE after inlining, compile with -Xprint:inliner -Xprint-icode. A textual .icode file is written in the output folder alongside .class files.

Listing 5: Shape B that triggers TCO rewriting

```scala
case Apply(fun, args) =>
  lazy val defaultTree = treeCopy.Apply(tree, fun, transformTrees(args, mkContext(ctx, false)));
  if (ctx.currentMethod.isFinal &&
    ctx.tailPos &&
    isRecursiveCall(fun)) {
    fun match {
      case Select(receiver, _) =>
        if (!forMSIL)
          rewriteTailCall(fun, receiver :: transformTrees(args, mkContext(ctx, false)))
        else
          defaultTree
      case _ => rewriteTailCall(fun, This(currentClass) :: transformTrees(args, mkContext(ctx, false)))
    }
  } else
    defaultTree
```
Figure 1: Inlining does take place, but at runtime the stack will grow anyway

4  Trampolines

Any discussion of TCO without trampolines and continuations would be incomplete. These notes are not complete, then! OK, not quite. An example about trampolines appears in Listing 7.

More on trampolines (of the scala.util.control.TailRec[A] variety) can be found here:


5  Continuations

As for continuations, a bunch of pointers can be found in a blog entry by Guy L. Steele Jr., “Why Object-Oriented Languages Need Tail Calls” [http://projectfortress.sun.com/Projects/Community/blog/ObjectOrientedTailRecursion](http://projectfortress.sun.com/Projects/Community/blog/ObjectOrientedTailRecursion).

Regarding Scala support for continuations:

- discussion at scala-lang.org, [http://www.scala-lang.org/node/2096](http://www.scala-lang.org/node/2096)
- the plugin, [http://www.scala-lang.org/node/2096](http://www.scala-lang.org/node/2096)

---

Listing 6: Example to show interplay between TCO and inlining

```scala
import annotation.tailrec

object tailRecExperiment5 {
  def main(args : Array[String]) {
    val v = 5; println("factorial \( * v \) = \( + \) factorial\( v \))
  }
  final def factorial(n: Int) = loop(n, 1)
  @inline final def justAnEnclosingMethod(n: Int, acc: Int): Int = loop(n, acc)
  @tailrec @inline final def loop(n: Int, acc: Int): Int =
    if (n <= 0) acc
    else if (n == 1) 1 * loop(n - 1, acc * n)
    else if (n == 2) justAnEnclosingMethod(n - 1, acc * n)
    else loop(n - 1, acc * n)
}
```

Listing 7: Mutually recursive invocations, trampoline style

```scala
package tailRecExperiment

sealed trait Bounce[A]
  case class Done[A](result: A) extends Bounce[A]
  case class Call[A](thunk: () => Bounce[A]) extends Bounce[A]

object tailRecExperiment2 extends Application {
  def even2(n: Int): Bounce[Boolean] = {
    if (n == 0) Done(true)
    else Call(() => odd2(n - 1))
  }
  def odd2(n: Int): Bounce[Boolean] = {
    if (n == 0) Done(false)
    else Call(() => even2(n - 1))
  }
  def trampoline[A](bounce: Bounce[A]): A = bounce match {
    case Call(thunk) => trampoline(thunk())
    case Done(x) => x
  }
  println(trampoline(even2(9999)))
}
```