Rewriting a method body to eliminate recursive tail calls

© Miguel Garcia, LAMP, EPFL
http://lamp.epfl.ch/~magarcia

September 26th, 2011

Abstract

The `tailcalls` phase gets its name from "Tail call elimination", the process of rewriting the body of a non-overridable method *m* (containing tail-recursive invocations to *m*, possibly on an instance different from `this`) into a loop, with tail-recursive callsites replaced by back-edges. Additionally, `tailcalls` shows a convenient way of processing nested evaluation contexts (a technique that simplifies AST processing a lot).

```
      phase name id description
      ---------- -- -----------
          parser  1 parse source into ASTs, perform simple desugaring
           namer  2 resolve names, attach symbols to named trees
  packageobjects  3 load package objects
           typer  4 the meat and potatoes: type the trees
  superaccessors  5 add super accessors in traits and nested classes
         pickler  6 serialize symbol tables
       refchecks  7 reference/override checking, translate nested objects
        liftcode  8 reify trees
         uncurry  9 uncurry, translate function values to anonymous classes
/*----------------------------------------------------------------------*/
       tailcalls 10 replace tail calls by jumps
/*----------------------------------------------------------------------*/
      specialize 11 @specialized-driven class and method specialization
   explicitouter 12 this refs to outer pointers, translate patterns
         erasure 13 erase types, add interfaces for traits
       lazyvals 14 allocate bitmaps, translate lazy vals into lazified defs
      lambdalift 15 move nested functions to top level
    constructors 16 move field definitions into constructors
         flatten 17 eliminate inner classes
           mixin 18 mixin composition
         cleanup 19 platform-specific cleanups, generate reflective calls
           icode 20 generate portable intermediate code
         inliner 21 optimization: do inlining
        closelim 22 optimization: eliminate uncalled closures
             dce 23 optimization: eliminate dead code
             jvm 24 generate JVM bytecode
        terminal 25 The last phase in the compiler chain
```

Contents

Listing 1: Sec. 1.1

```
class C(that: C) {

  def factorial(n: Int) = tcfact(1, n)

  private def tcfact(acc: Int, n: Int): Int =
    if(n == 0)
      acc
    else if(n == 1) {
      val nonTail = tcfact(acc * n, n - 1)
      nonTail
    } else if(n == 2) {
      that.tcfact(acc * 2, 1)
    } else
      tcfact(acc * n, n - 1);

}
```

Listing 2: Sec. 1.1

```
private def tcfact(acc: Int, n: Int): Int = {
  <synthetic> val _$this: C = C.this;
  _tcfact(_$this,acc,n){
    if (n.==(0))
      acc
    else
      if (n.==(1))
        {
          val nonTail: Int = C.this.tcfact(acc.*(n), n.-(1));
          nonTail
        }
      else
        if (n.==(2))
          _tcfact(C.this.that, acc.*(2), 1)
        else
          _tcfact(C.this, acc.*(n), n.-(1))
  }
}
```

1 Intro

1.1 Shape of the transformed AST

The program in Listing 1 contains recursive invocations (targeting `tcfact()`) in both tail and non-tail-position, in one case with a receiver different from `this`. The snippet in Listing 2 shows the resulting AST, with a method-level `_$this` variable and a loop. Callsites in non-tail positions remain as-is.

Some comments:

- had the method been annotated `@tailrec`, any non-rewritable recursive call leads to compile error.

  ```
  def isMandatory    = method.hasAnnotation(TailrecClass) && !forMSIL
  ```

- the full story about a method being non-overridable is:

  ```
  /** Is this symbol effectively final? I.e, it cannot be overridden */
  final def isEffectivelyFinal: Boolean = (
      isFinal
    || hasModuleFlag && !settings.overrideObjects.value
    || isTerm && (
          isPrivate
        || isLocal
        || owner.isClass && owner.isEffectivelyFinal
    )
  )
  ```

1.2 Connection to other phases

`tailcalls` runs early in the pipeline: multiple parameter lists have been collapsed into a single one (by `uncurry`) but other than that ASTs are pretty rich at this point: input ASTs may contain local definitions, outer-inner classes, traits, and so on. However `tailcalls` can do its work just by considering a single method at a time.

In turn, follow-up phases do not depend on taill-call elimination. In fact the transform is deactivated by choosing the `notailcalls` debug level:

```
  val g = ChoiceSetting ("-g", "level", "Set level of generated debugging info.",
                       List("none", "source", "line", "vars", "notailcalls"),
                       "vars")
```

2 Mechanics

2.1 Passing context down the tree

Imagine you're the `TailCallElimination` transformer. Upon visiting a callsite that looks promising (i.e. it's recursive, to an eligible method) how do you know whether it's in tail-position? The answer is given by the "current context", which is well-defined within a method. Whenever a new context is established, the previous one is re-instantiated upon returning from the transformation under the new context:

```
def transform(tree: Tree, nctx: Context): Tree = {
  val saved = ctx
  ctx = nctx
  try transform(tree)
  finally this.ctx = saved
}
```

The following two ways are representative of establishing the current context:

1. Upon visiting any method:

   ```
   case dd @ DefDef(mods, name, tparams, vparams, tpt, rhs) =>
     val newCtx = new Context(dd)
     val newRHS = transform(rhs, newCtx)
     . . .
   ```

2. To mark some sub-expressions as not being in tail-position. For example: "*no calls inside a try are in tail position, but keep recursing for nested functions*"

   ```
   case Try(block, catches, finalizer) =>
      // no calls inside a try are in tail position, but keep recu
     treeCopy.Try(tree,
       noTailTransform(block),
       noTailTransforms(catches).asInstanceOf[List[CaseDef]],
       noTailTransform(finalizer)
     )
   ```

   In words, `noTailTransform(tree)` transforms `tree` under the influence of `noTailContext()` as current context (which amounts to a copy of the current context, save for its `tailPos` flag which indicates "currently not in tail-position").

2.2 Querying the context on the way back

In some cases, the visitor needs to know what happened downstream. For example, whether all eligible callsites were actually turned into jumps (as required by "`@tailrec`"). In the example, those rewritings also set the `accessed` field on the visitor:

```
def rewriteTailCall(recv: Tree): Tree = {
  log("Rewriting tail recursive call: " + fun.pos.lineContent.trim)

  ctx.accessed = true
  typedPos(fun.pos)(Apply(Ident(ctx.label), recv :: transformArgs))
}
```

Afterwards, upon leaving a `DefDef` node, the context can be queried:

```
treeCopy.DefDef(tree, mods, name, tparams, vparams, tpt, {
  if (newCtx.isTransformed) {
    /** We have rewritten the tree, but there may be nested recursive calls remaining.
     *  If @tailrec is given we need to fail those now.
     */
    if (newCtx.isMandatory) {
      for (t @ Apply(fn, _) <- newRHS ; if fn.symbol == newCtx.method) {
        newCtx.failPos = t.pos
        newCtx.tailrecFailure()
      }
    }
```

2.3 Under the hood

The previous sections already cover the main points about the transformation. The method shown in Listing 3 conveys most of the remaining details. It's invoked as shown below (notice the special casing of short-circuit evaluation):

```
case Apply(tapply @ TypeApply(fun, targs), vargs) =>
  rewriteApply(tapply, fun, targs, vargs)

case Apply(fun, args) =>
  if (fun.symbol == Boolean_or || fun.symbol == Boolean_and)
    treeCopy.Apply(tree, fun, transformTrees(args))
  else
    rewriteApply(fun, fun, Nil, args)
```

Listing 3: Sec. 2.3

```
/** A possibly polymorphic apply to be considered for tail call transformation.
 */
def rewriteApply(target: Tree, fun: Tree, targs: List[Tree], args: List[Tree]) = {
  val receiver: Tree = fun match {
    case Select(qual, _) => qual
    case _               => EmptyTree
  }

  def receiverIsSame  = ctx.enclosingType.widen =:= receiver.tpe.widen
  def receiverIsSuper = ctx.enclosingType.widen <:< receiver.tpe.widen
  def isRecursiveCall = (ctx.method eq fun.symbol) && ctx.tailPos
  def transformArgs   = noTailTransforms(args)
  def matchesTypeArgs = ctx.tparams sameElements (targs map (_.tpe.typeSymbol))

  /** Records failure reason in Context for reporting.
   *  Position is unchanged (by default, the method definition.)
   */
  def fail(reason: String) = {
    debuglog("Cannot rewrite recursive call at: " + fun.pos + " because: " + reason)

    ctx.failReason = reason
    treeCopy.Apply(tree, target, transformArgs)
  }
  /** Position of failure is that of the tree being considered.
   */
  def failHere(reason: String) = {
    ctx.failPos = fun.pos
    fail(reason)
  }
  def rewriteTailCall(recv: Tree): Tree = {
    log("Rewriting tail recursive call: " + fun.pos.lineContent.trim)

    ctx.accessed = true
    typedPos(fun.pos)(Apply(Ident(ctx.label), recv :: transformArgs))
  }

  if (!ctx.isEligible)        fail("it is neither private nor final so can be overridden")
  else if (!isRecursiveCall) {
    if (receiverIsSuper)      failHere("it contains a recursive call targetting a supertype")
    else                      failHere(defaultReason)
  }
  else if (!matchesTypeArgs)  failHere("it is called recursively with different type arguments")
  else if (receiver == EmptyTree) rewriteTailCall(This(currentClass))
  else if (forMSIL)           fail("it cannot be optimized on MSIL")
  else if (!receiverIsSame)   failHere("it changes type of 'this' on a polymorphic recursive call")
  else                        rewriteTailCall(receiver)
}
```