Tail-call optimization

Compiler Construction 2014 - Final Report

Loris Leiva Cyriaque Brousse

EPFL

1. Introduction

After writing the lexer, which divides source code into individual tokens, the parser, which creates the abstract syntax tree from the tokens previously generated, we needed to implement code-checking tools, namely the name analyzer that rejects source code containing errors on names (e.g. an undeclared variable is used), and the type checker, which enforces type constraints on the program. We were then sure that all invalid programs were rejected. Finally, the code generation phase outputs Java bytecodes from the abstract syntax tree, which are further translated by the virtual machine, then executed.

The aim of our compiler extension was to allow developers to use multiple return statements within a same method, and eliminate tail calls, thus use less stack space.

These new language constructs called for some modifications of the features previously written.

2. Examples

Our program `TailRecursive.tool`, of which an excerpt can be found below, illustrates well our purpose: we want to sum the elements of an integer linked list. The implementation of the `Node` class is straightforward and therefore not shown here.

```
def sumList(head: Node): Int = {
    return this.sumLoop(head, 0);
}

def sumLoop(node: Node, accumulator: Int): Int = {
    if (node.hasNext()) {
        return this.sumLoop(node.next(),
            node.getValue() + accumulator);
    } else {
        return accumulator;
    }
}
```

We can see the two return statements in the $sumLoop$ method. Also, this method uses tail-recursion with the accumulator. We want to eliminate this tail call by updating the value of the accumulator and going back to the beginning of the method. By this mean we get rid of the recursion.

Our modifications shall output the following bytecodes for these two methods:

```
public int sumList(Node);
  Code:
     0: aload_0
     1: aload_1
     2: iconst_0
     3: invokevirtual #77 // Method sumLoop:(LNode;I)I
     6: ireturn

public int sumLoop(Node, int);
  Code:
     0: aload_1
     1: invokevirtual #80 // Method Node.hasNext:()Z
     4: ifeq 22
     7: aload_1
     8: invokevirtual #83 // Method Node.next:()LNode;
    11: astore_1
    12: aload_1
    13: invokevirtual #87 // Method Node.getValue:()I
    16: iload_2
    17: iadd
    18: istore_2
    19: goto 0 // tail-call is eliminated
    22: iload_2
    23: ireturn
```

3. Implementation

3.1 Theoretical Background

In order to eliminate tail calls, we said we needed to update the function's arguments and resume its execu-

tion at its beginning (thus with the new parameters). Therefore, each function needs to be assigned a label.

Since the `return` is now a regular statement, the regular expression of the body of the method (specified in the method declaration) needs to be changed from

{ (VarDeclaration)∗ (Statement)∗ **return** Expression ; }

to simply

{ (VarDeclaration)∗ (Statement)∗ }

The name analyzer for statements needs to be improved to check whether the function actually terminates (since the `return` statement is now part of a statement block, not on its own at the end of the method in general. For instance, the *if* statement

if (cond) stat1 **else** stat2

terminates if and only if both `stat1` and `stat2` terminate (contain a return statement). Let us now consider a second example. Consider the following statement:

if (cond) stat1

Even if `stat1` terminates, we are not guaranteed that the whole *if* statement terminates (in the case `cond` evaluates to false). Therefore we need a return statement among the following statements of the same block.

Also, the Java compiler forbids programmers to write code after a return statement. Indeed, when one tries to do so, an error is displayed (not a warning). So we figured we should output an error as well and abort compilation if this is the case in a Tool program.

3.2 Implementation Details

We will discuss each compiler phase successively.

- As the new regular expression for the method declaration denotes, we needed to create a new `StatTree` for the return statement and remove it from the `MethodDecl` tree.
- The parser had to be modified accordingly so that we consider the ”return” case, and that the method declaration doesn't ”eat” a return expression anymore (i.e. expects a said token, moves to the next one if it finds it, or outputs an error if it does not).
- In the name analyzer, we needed not to browse for a return expression in the case of a method declaration anymore, but rather make a ”return” case of its own. It is still not allowed to return from the main method, which is specified there.

To address the problem of a statement that might or might not terminate, as explicited in the previous subsection, we decided to add a boolean variable *terminal* to the `StatTree` trait. We then recursively descend the abstract syntax tree to check whether each method has the right amount of return statements, wherever they are expected. The special case of an *if* statement we explained is handled in the `isTerminal` method of step 4 in the code, where we check that both the then and the else branch actually contain a return statement.

```
def isTerminal(t: StatTree): Boolean = t match {
    case Block(stats) =>
        var terminal = false
        stats foreach { s =>
            if (terminal)
                fatal("Unreachable code", s)
            if (isTerminal(s))
                terminal = true;
        }
        t.terminal = terminal
        terminal

    case If(expr, thn, els) => els match {
        case Some(e) =>
            val b = isTerminal(thn) && isTerminal(e)
            t.terminal = b
            b
        case None => false
    }

    // final value
    case _ => t.terminal
}
```

The default values of `terminal` are defined in `Tree.scala`

```
sealed trait StatTree extends Tree {
    var terminal : Boolean = false
}
// ...
case class Return(expr: ExprTree) extends StatTree {
    terminal = true
}
```

The case of unreachable code is also handled in the name analyzing phase, namely in the `checkTerminals` and `isTerminal` methods. There you can see that

we verify for each statement block that no statement follows a return statement.

```
def checkReturnStatements(prog : Program): Unit =
{
  prog.classes foreach { c =>
    c.methods foreach { m =>
      val methodIsTerminal = checkTerminals(m.stats)
        if (!methodIsTerminal)
          fatal("return statements missing", m)
    }
  }
}

def checkTerminals(stats: List[StatTree]) : Boolean =
{
  var terminal = false

  stats foreach { s =>
    if (terminal)
      fatal("Unreachable code", s)

    if (isTerminal(s))
      terminal = true;
  }
  terminal
}
```

- Our extension did not call for theoretical changes in the type checker, which made the implementation modifications rather straightforward. We needed to once again pay attention to the fact that there is now a proper return statement. As you can see in the code, we now distinguish between a return statement and a regular statement.

```
def tcStat(stat: StatTree)(implicit scope : Symbol):
    Unit = stat match
{
    //...

    case Return(retExpr) =>
        tcExpr(retExpr, scope.getType)
}
```

- The code generation phase modifications were the most intricate ones. We first differentiated between a return statement and a regular statement, which is nothing new compared to the previous phases. Then, we eliminated tail calls. We first need to detect such a call, which we did by matching the return expression against a method call, and further matching the object on which the method is called against the self reference (*this*). If the called method is the same as the one we are currently in, then it is a tail call. In this case, we zip the method argument list with the method call's. Then, for each argument of this list, we update the corresponding value, by means of the bytecodes `ISTORE` (for integers and booleans) and `ASTORE` for every other types.

```
retExpr match {
  case MethodCall(obj, meth, args) => obj match {
    case This() =>
      if (meth.getSymbol equals m) {
        val argsAndNewValues = m.argList zip args
        argsAndNewValues foreach { case (v,e) =>
          compile(e)
          v.getType match {
            case TInt|TBoolean =>
              ch << IStore(slots.slotFor(v.name))
            case _ =>
              ch << AStore(slots.slotFor(v.name))
          }
        }
        ch << Goto(startLabel)
      } else {
        compileReturn
      }
    case _ => compileReturn
  }
  case _ => compileReturn
}
```

Where the function `compileReturn` compile the return statement normally (i.e. no tail-call recursion found).

```
def compileReturn() = {
  compile(retExpr)
  m.getType match {
    case TInt|TBoolean => ch << IRETURN
    case _ => ch << ARETURN
  }
}
```

4. Possible Extensions

Since our extension is about tail-call elimination, it can hardly be subject to any further improvements. We could, though, eliminate tail calls in imbricated functions, such as in Scala.

We particularly liked this lab, since it made use rebrand all the steps of our compiler, and makes Tool a step closer to Scala.