GOTO elimination for Scala ASTs

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

February 10th, 2011

Abstract

Unparsing, translations (for example into JavaScript or C# 3.0) and many other tasks have a hard time dealing with the jumps introduced by `TransMatch`. We want to develop a custom compiler phase that rephrases those AST subtrees (in a semantics-preserving manner) for reuse by other compiler plugins (such as the unparser and backends targeting `goto`-less languages). Previous work on *GOTO elimination* does result in some (minor) performance degradation for programs with high GOTO density. This has to be weighed against the additional optimizations applicable to programs exhibiting structured control flow. We consider the algorithm devised by Erosa and Hendren [2], as well as the changes required to manipulate Scala ASTs.

Contents

1 How to build and run **3**

2 Background **3**
- 2.1 Problem statement . . . 3
- 2.2 Running example after GOTO elimination . . . 3
- 2.3 Theory on GOTO statement elimination . . . 4
- 2.4 Sidenote: breaking with the habit of C's `break` and `continue` . . 4
- 2.5 What about backends for languages with "structured-GOTO" . . 5

3 AST shapes of interest **5**
- 3.1 Non-problematic `LabelDef-Apply` shapes . . . 5
- 3.2 How `GenICode` translates `LabelDef`s and jumps . . . 6
- 3.3 Those `Match` nodes surviving `TransMatch` have `switch` semantics 8
- 3.4 `try-catch-finally` . . . 8

4 Before GOTO elimination **9**
- 4.1 `LabelDef-Apply` in Scala vs. `goto-label` in C . . . 9
- 4.2 Block flattening . . . 10
- 4.3 Checking of preconditions . . . 11
- 4.4 Classifying `LabelDef-Apply` pairs . . . 12
- 4.5 Checking postconditions . . . 13
- 4.6 Gaining details about tree shapes . . . 13
- 4.7 In terms of our running example . . . 15

5 Adapting the Erosa-Hendren algorithm to Scala **16**
- 5.1 Transform for sibling pairs . . . 16
- 5.2 Outward movement transform . . . 16
- 5.3 Inward movement transform . . . 16
- 5.4 Putting it all together . . . 16

6 Related Work **17**
- 6.1 Decompiling Scala into Java . . . 17
- 6.2 Translating Scala ASTs into JavaScript . . . 17

7 Future Work **17**
- 7.1 A new backend for the Scala.Net compiler to emit C# sources . . 17

1 How to build and run

1. compile all Scala source files from `http://lampsvn.epfl.ch/trac/scala/browser/scala-experimental/trunk/gotoelim`
2. say the resulting classfiles are found at `myplugins\gotoelim\classes`
3. prepare the `gotoelim.jar` as follows

```
del gotoelim.jar
jar -cf gotoelim.jar -C myplugins\gotoelim\classes scala -C myplugins\gotoelim\resources\ .
```

4. where `myplugins\gotoelim\resources` contains the plugin manifest `scalac-plugin.xml`

```
<plugin>
       <name>gotoelim</name>
       <classname>scala.tools.gotoelim.GotoElimPlugin</classname>
</plugin>
```

Afterwards, just run `scalac` with `-Xplugin where/to/find/gotoelim.jar`.

2 Background

2.1 Problem statement

The issue has been reported a number of times[1].

```
class YouBadPatternMatcher {
  def problematicPattern = {
    try {
      0
    } catch {
      case x: Exception if x.getMessage == "test" => println("first case " + x)
      case x => println("second case " + x)
    }
  }
}
```

After the `cleanup` phase, the fragment above is reduced as shown in Listing 1 (to reproduce, run the compiler with `-uniqid -Xprint:cleanup`).

Please notice that the jump in question can also not be expressed in languages with "structured `goto`" (such as C#) because the target instruction is not directly contained in an outer block (but two blocks deep inside the nearest common block for jump source and target).

2.2 Running example after GOTO elimination

Our plugin should rephrase the program from Listing 1 as shown in Listing 3 and Listing 4.

[1] `http://www.scala-lang.org/node/7423`

2.3 Theory on GOTO statement elimination

- Ana M. Erosa and Laurie J. Hendren. *Taming Control Flow: A Structured Approach to Eliminating GOTO Statements*[2] In ICCL, 1994.
- Todd A. Proebsting and Scott A. Watterson.
  Krakatoa: Decompilation in Java (does bytecode reveal source?)[3]

GOTO removal is popular when "levitating" legacy code into some "higher-level" form:

- `http://selab.fbk.eu/ceccato/papers/2008/csmr2008.html`
- `http://home.comcast.net/~refilman/text/invision/nogo.pdf`

Other general references:

- `http://www.program-transformation.org/Transform/JavaDecompilers`
- `http://www.sable.mcgill.ca/dava/`

2.4 Sidenote: breaking with the habit of C's `break` and `continue`

Quoting from the Scala FAQ, `http://www.scala-lang.org/node/257`:

> *These keywords are not included in Scala 2.7, and must be implemented in a different way. For* `break`*, the simplest thing to do is to divide your code into smaller methods and use the return to exit early. For* `continue`*, a simple approach is to place the skipped-over parts of a loop into an* `if`*.*
>
> *Scala 2.8 will include* `break`*, but not* `continue`*.*

What Scala 2.8 includes are library abstractions in `scala.util.control`. Discussions in reverse chronological order:

- `http://www.scala-lang.org/node/3638`
- `http://www.scala-lang.org/node/2065`
- `http://www.scala-lang.org/node/1792`
- `http://www.scala-lang.org/node/1229`

[2] `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.54.9516`

[3] `http://www.usenix.net/publications/library/proceedings/coots97/full_papers/proebsting2/proebsting2.pdf`

2.5 What about backends for languages with "structured-GOTO"

GOTO elimination is also necessary for (future) backends (Sec. 7.1) targeting languages with "structured-GOTO". For example, C# does include a `goto` statement, but basically only allowing inner-to-outer control transfers. Quoting from the C# 3.0 spec:

> *The target of a `goto identifier` statement is the labeled statement with the given label. If a label with the given name does not exist in the current function member, or if the `goto` statement is not within the scope of the label, a compile-time error occurs. This rule permits the use of a `goto` statement to transfer control out of a nested scope, but not into a nested scope.*
>
> *The target of a `goto case` statement is the statement list in the immediately enclosing `switch` statement (§8.7.2), which contains a `case` label with the given constant value. If the `goto case` statement is not enclosed by a `switch` statement, if the constant-expression is not implicitly convertible (§6.1) to the governing type of the nearest enclosing `switch` statement, or if the nearest enclosing `switch` statement does not contain a `case` label with the given constant value, a compile-time error occurs.*
>
> *The target of a `goto default` statement is the statement list in the immediately enclosing `switch` statement (§8.7.2), which contains a `default` label. If the `goto default` statement is not enclosed by a `switch` statement, or if the nearest enclosing `switch` statement does not contain a `default` label, a compile-time error occurs.*
>
> *A `goto` statement cannot exit a `finally` block (§8.10). When a `goto` statement occurs within a `finally` block, the target of the `goto` statement must be within the same `finally` block, or otherwise a compile-time error occurs.*

3 AST shapes of interest

3.1 Non-problematic `LabelDef-Apply` shapes

Some `LabelDef-Apply` shapes denote `do-while` and `while` loops. Their implied GOTOs are not going to be eliminated. The snippet below detects them, also in case the "`goto` part" was unnested from its block (*block flattening* is covered in Sec. 4.2). In the example unnesting occurs if the block contains just that `Apply` (of the form `Apply(Ident, List())`), i.e. for an infinite loop as in `while(true){}`.

```
def isWhileLoop(tree: LabelDef): Boolean = tree match {

  // while (cond) body ==> LabelDef($L, List(), if (cond) { body; L$() } else ())
  case lab @ LabelDef(name, List(),
                  If(cond, thenp @ Block(stats, laCall @ Apply(laCallFun : Ident, List())), _))
      if(lab.symbol eq laCallFun.symbol) => true

  // this case results from BlockFlattening
  case lab @ LabelDef(name, List(),
```

```
case LabelDef(name, params, rhs) =>
  val ctx1 = ctx.newBlock
  if (nme.isLoopHeaderLabel(name))
    ctx1.bb.loopHeader = true

  ctx1.labels.get(tree.symbol) match {
    case Some(label) =>
      log("Found existing label for " + tree.symbol)
      label.anchor(ctx1.bb)
      label.patch(ctx.method.code)

    case None =>
      val pair = (tree.symbol -> (new Label(tree.symbol) anchor ctx1.bb setParams (params map (_.symbol))
      log("Adding label " + tree.symbol + " in genLoad.")
      ctx1.labels += pair
      ctx.method.addLocals(params map (p => new Local(p.symbol, toTypeKind(p.symbol.info), false)));
  }

  ctx.bb.closeWith(JUMP(ctx1.bb), tree.pos)
  genLoad(rhs, ctx1, expectedType /*toTypeKind(tree.symbol.info.resultType)*/)
```

Figure 1: Sec. 3.2

```
                    If(cond, laCall @ Apply(laCallFun : Ident, List()), _))
      if(lab.symbol eq laCallFun.symbol) => true

  case _ => false
}
```

```
def isDoWhileLoop(tree: LabelDef): Boolean = tree match {

  // do body while (cond) ==> LabelDef($L, List(), body; if (cond) L$() else ())
  case lab @ LabelDef(_, List(), Block(stats, If(cond, laCall @ Apply(laCallFun : Ident, List()), _)))
      if(lab.symbol eq laCallFun.symbol) => true

  // this case results from BlockFlattening
  case lab @ LabelDef(_, List(), If(cond, laCall @ Apply(laCallFun : Ident, List()), _))
      if(lab.symbol eq laCallFun.symbol) => true

  case _ => false
}
```

3.2 How `GenICode` translates `LabelDefs` and jumps

The translation into ICode of `LabelDef` is shown in Figure 1 and that of a jumping `Apply` in Figure 2.

A `LabelDef` receives arguments (and a jumping `Apply` provides them) which amounts to assigning method-local vars at the point of jump (assignments effected by ICode `STORE_THIS` and `STORE_LOCAL` instructions), as shown below.

These method-local vars present no problem for GOTO elimination, because:

- they aren't used for any purpose other than communicating values from the source of the jump to its destination; and
- the transformed program follows the same execution paths (for similar inputs) as before transformation. That means, for each execution path

```
case app @ Apply(fun, args) =>
  val sym = fun.symbol

  if (sym.isLabel) {  // jump to a label
    val label = ctx.labels.getOrElse(sym, {
      // it is a forward jump, scan for labels
      log("Performing scan for label because of forward jump.")
      scanForLabels(ctx.defdef, ctx)
      ctx.labels.get(sym) match {
        case Some(l) =>
          log("Found label: " + l)
          l
        case _       =>
          abort("Unknown label target: " + sym +
                " at: " + (fun.pos) + ": ctx: " + ctx)
      }
    })
    val ctx1 = genLoadLabelArguments(args, label, ctx)
    ctx1.bb.emitOnly(if (label.anchored) JUMP(label.block) else
    ctx1.bb.enterIgnoreMode
    ctx1
  } else if (isPrimitive(sym)) { // primitive method call
```

Figure 2: Sec. 3.2

reaching a `LabelDef`, the same initialization (or lack thereof) as in the original program will have occurred for those variables.

```
/**
 * Generate code that loads args into label parameters.
 */
private def genLoadLabelArguments(args: List[Tree], label: Label, ctx: Context): Context = {
  if (settings.debug.value) {
    assert(args.length == label.params.length,
           "Wrong number of arguments in call to label " + label.symbol)
  }

  var ctx1 = ctx

  def isTrivial(kv: (Tree, Symbol)) = kv match {
    case (This(_), p) if p.name == nme.THIS => true
    case (arg @ Ident(_), p) if arg.symbol == p => true
    case _                                  => false
  }

  val stores = args zip label.params filterNot isTrivial map {
    case (arg, param) =>
      val local = ctx.method.lookupLocal(param).get
      ctx1 = genLoad(arg, ctx1, local.kind)

      val store =
        if (param.name == nme.THIS) STORE_THIS(toTypeKind(ctx1.clazz.symbol.tpe))
        else STORE_LOCAL(local)

      store setPos arg.pos
  }

  // store arguments in reverse order on the stack
```

```
  ctx1.bb.emit(stores.reverse)
  ctx1
}
```

3.3 Those `Match` nodes surviving `TransMatch` have `switch` semantics

```
  /**
   *   Pattern matching expression (before explicitouter)
   *   Switch statements          (after explicitouter)
   *
   *
   *   After explicitouter, cases will satisfy the following constraints:
   *
   *   - all guards are EmptyTree,
   *   - all patterns will be either Literal(Constant(x:Int))
   *     or Alternative(lit|...|lit)
   *     except for an "otherwise" branch, which has pattern
   *       Ident(nme.WILDCARD)
   */
  case class Match(selector: Tree, cases: List[CaseDef])
       extends TermTree
```

Please glean the rest of the meaning of `Match` nodes from the `GenICode` snippet shown in Listing 2 on p. 19 and from the following snippet for ICode's `SWITCH` instruction.

```
/** This class represents a SWITCH instruction
 * Stack: ...:index(int)
 *   ->: ...:
 *
 * The tags array contains one entry per label, each entry consisting of
 * an array of ints, any of which will trigger the jump to the corresponding label.
 * labels should contain an extra label, which is the 'default' jump.
 */
case class SWITCH(tags: List[List[Int]], labels: List[BasicBlock]) extends Instruction {
  /** Returns a string representation of this instruction */
  override def toString(): String ="SWITCH ..."

  override def consumed = 1
  override def produced = 0
}
```

3.4 `try-catch-finally`

At the level of Scala sources, occurrences of `try-catch`, `try-finally`, and `try-catch-finally` are possible. The spec states that the third pattern is semantically equivalent to a `finally` protecting a `try-catch` expression. However this semantic equivalence is not reflected in ASTs, where `try-catch-finally` is not represented as two nested try statements but as a single node:

```
  case class Try(block: Tree, catches: List[CaseDef], finalizer: Tree)
       extends TermTree
```

The desugaring of try-catch-finally can only be seen at `GenICode` time. Quoting from `GenICode.scala`:

```
/** try-catch-finally blocks are actually simpler to emit in MSIL, because there
 *  is support for `finally` in bytecode.
 *
 *  A
 *    try { .. } catch { .. } finally { .. }
 *  block is de-sugared into
 *    try { try { ..} catch { .. } } finally { .. }
 *
 *  In ICode `finally` block is represented exactly the same as an exception handler,
 *  but with `NoSymbol` as the exception class. The covered blocks are all blocks of
 *  the `try { .. } catch { .. }`.
 *
 *  Also, TryMsil does not enter any Finalizers into the `cleanups`, because the
 *  CLI takes care of running the finalizer when seeing a `leave` statement inside
 *  a try / catch.
 */
def TryMsil(body: Context => Context,
            handlers: List[(Symbol, TypeKind, (Context => Context))],
            finalizer: Tree,
            tree: Tree) = {
```

4 Before GOTO elimination

Erosa and Hendren [2] devised an algorithm to transform a C program containing arbitrary GOTO statements into a version where only `break` and `continue` are used in addition to `if/else`, while-loop, and `switch`. In this section, we gather information in order to answer (in Sec. 5) the following questions:

1. which of the original transformations in [2] are applicable as-is to Scala ASTs (maybe a subset of all transforms?); and
2. which changes are needed to avoid introducing `break` and `continue`.

Regarding the interplay of GOTO elimination with:

- `try-catch-finally`: Jumps are introduced only for `match` expressions, thus jumps are not only intra-method but moreover always enclosed within some expression (thus, a jump cannot straddle between a protected block in a `try` clause to a `case` clause in a `catch`, for example).
- runtime exceptions (e.g., division by zero): the transformation we'll apply does not alter the lexical nesting of `try-catch-finally`, and thus preserves exceptional behavior, in particular upon exceptions thrown implicitly by the VM.

4.1 `LabelDef-Apply` in Scala vs. `goto-label` in C

A `LabelDef-Apply` pair denotes two tree nodes, where the symbol of the `LabelDef` indicates it's the target of the jump given by the `Apply`. More than one jump may target the same `LabelDef`.

We start the development of the `gotoelim` compiler plugin by writing the code in charge of reporting which tree nodes require transformation. Based on the classification of `goto-label` pairs in [2], we classify `LabelDef-Apply` pairs into:

- `do-while` and `while` loops: these `LabelDef-Apply` pairs are left as-is.
- "sibling" pairs.
- "directly related" pairs, i.e., cases 3(a) and 3(b) in [2].
- "indirectly related", i.e., cases 3(c) and 3(d) in the paper. Difference between them: the nearest enclosing statement `stmt_k` is a block in 3(c) as opposed to an `if` or `switch` in 3(d). In terms of the paper's notation, `stmt_k` is a composite statement that encloses (directly or indirectly) `stmt_i` and `stmt_j`.

Some pre- and post-conditions for the classification above:

- Precondition (1 of 3): each jump should have a matching `LabelDef`.
- Precondition (2 of 3): A `LabelDef` not targeted by any jump should have no `params`, in which case it can be replaced by its `rhs`.
- Precondition (3 of 3): Regarding `try-catch-finally`, jumps do not straddle protected regions.
- Postcondition: For a given Scala program P, any of the categories above may be empty, but every `LabelDef-Apply` pair in the AST of P should belong to exactly one of the categories above.

When populating the categories above, AST visiting could be kept to a minimum in case `TransMatch` would tell us which trees correspond to rewritten `Match` nodes (the jumps we want to eliminate don't appear anywhere else). Failing that, we have to visit all outermost expressions.

We get back to the classification procedure, and its pre and postconditions in Sec. 4.3 ff., but before we have to discuss block flattening.

4.2 Block flattening

The algorithm we'll use to eliminate GOTOs relies on the notion of *statement sequence*, where the only composite statements are loops (their body is a statement sequence), `if/else`, and `switch`. Additionally, a C procedure body is also a statement sequence.

In our case we have other composite statements (e.g., `try-catch-finally`) but we can get closer to the expected formulation of "statement sequences" by unnesting those statements in a block contained in another block. We call this *block flattening*.

The tree-rewriting part of block flattening is not difficult to get right (one can get inspiration from `allStatements` in `CompactTreePrinter`) but initially nothing worked, because I had forgotten to "`transformInfo`" (which amounts to identity when unnesting a block):

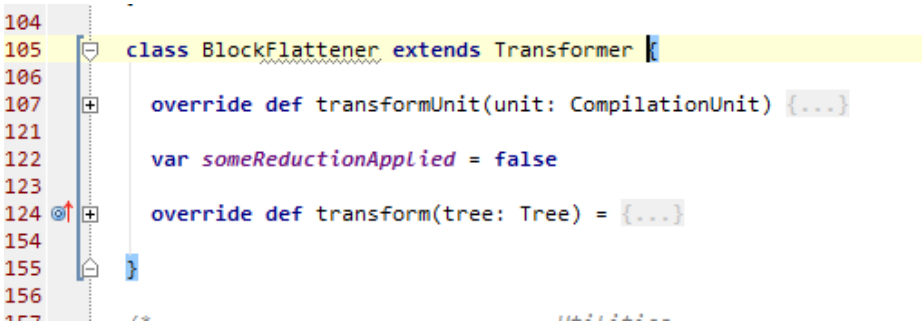
```
trait BlockFlattening extends nsc.SubComponent with nsc.transform.InfoTransform {
  /*- InfoTransform IS IMPORTANT */

  import global._
  import definitions._
  import typer.{typed, atOwner} // methods to type trees

  override def changesBaseClasses = false
  def transformInfo(sym: Symbol, tp: Type): Type = { tp } /*- THIS IS IMPORTANT TOO */

  . . .
```

Flattening is done iteratively (in `BlockFlattener.transformUnit`, until `someReductionApplied == false`), with each iteration performed by `BlockFlattener.transform`:




```
override def transform(tree: Tree) = tree match {

       case Block(List(), expr) =>
         assert (expr != EmptyTree)
         someReductionApplied = true;
         typed { transform(expr) }

       case Block(stats1, b2 @ Block(stats2, expr2)) =>
         someReductionApplied = true;
         val res0 = treeCopy.Block(tree, transformTrees(stats1 ::: stats2), transform(expr2))
         typed { res0 }

       case Block(stats, expr) if (stats exists (s => s.isInstanceOf[Block])) =>
         someReductionApplied = true;
         val (befBlock, blockEtc) = stats span (s => !s.isInstanceOf[Block])
         val Block(stats2, expr2) = blockEtc.head
         val newStats = befBlock ::: stats2 ::: List(expr2) ::: blockEtc.tail
         val res0 = treeCopy.Block(tree, transformTrees(newStats), transform(expr))
         typed { res0 }

       case _ => super.transform(tree)

}
```

4.3 Checking of preconditions

We check the preconditions listed in Sec. 4.1 as follows (at the beginning and end of the snippet below):

```
/* -------- Step (1) Flatten, make tidy, and report well-formedness errors -------- */

// the .scala filename is given by unit.source.file.path
new BlockFlattener transformUnit(unit)
// well-formedness checking and reporting
val glColl = new GotoLabelCollector apply unit.body
glColl.reportErrors()

// replace arg-less LabelDefs not targeted by any goto with their rhs
val labelsToGoAway = glColl.labelsLackingGoto map { lab => lab.symbol }
if(labelsToGoAway.nonEmpty) {
  new Label2Rhs(labelsToGoAway) transformUnit(unit)
  new BlockFlattener transformUnit(unit)
}

// check no jump straddles a region protected by exception-handling
new ExceptionStraddling traverse unit.body

// check that all Match nodes have switch semantics.
new AllMatchesAreSwitches traverse unit.body
```

where for example:

```
class ExceptionStraddling extends Traverser {

  override def traverse(tree: Tree) = {
    tree match {
      case tt: Try =>
        checkAllJumpsInternal(tt.block)
        for(c <- tt.catches) {
          // it's ok for a jump to leave the CaseDef.guard (I guess)
          checkAllJumpsInternal(c.body)
          // actually the above also prevents jumps from a CaseDef's body to that same CaseDef's guard
        }
        checkAllJumpsInternal(tt.finalizer)
      case _ => ()
    }
    super.traverse(tree)
  }

}
```

4.4 Classifying `LabelDef-Apply` pairs

This step allows visualizing what trees will be the focus of transformations later. To recap, the categories of interest (Sec. 4.1) are:

- `do-while` and `while` loops: these `LabelDef-Apply` pairs are left as-is.
- "sibling" pairs.
- "directly related" pairs, i.e., cases 3(a) and 3(b) in [2].
- "indirectly related", i.e., cases 3(c) and 3(d) in the paper. Difference between them: the nearest enclosing statement `stmt_k` is a block in 3(c) as opposed to an `if` or `switch` in 3(d). In terms of the paper's notation, `stmt_k` is a composite statement that encloses (directly or indirectly) `stmt_i` and `stmt_j`.

```
/* ---------------- Step (2) Classify trees of interest ---------------- */

/* (2.1) while and do-while loops */
gL = new GotoLabelCollector(unit.body)
for(lab <- gL.whileLoops) { gL.warningMultiJump(lab) }
for(lab <- gL.doWhileLoops) { gL.warningMultiJump(lab) }

/* (2.2) g-l and l-g siblings */
val siColl = new SiblingsCollector(unit.body)
siColl.reportInfo()

/* (2.3) directly related pairs */
val dirColl = new DirectRelCollector(gL, unit.body)
dirColl.reportInfo()

/* (2.4) indirectly related pairs */
val indiColl = new IndiRelCollector(gL, unit.body)
```

4.5 Checking postconditions

- Postcondition: For a given Scala program P, any of the categories (loop, sibling, directly related, indirectly related) may be empty, but every `LabelDef-Apply` pair in the AST of P should belong to exactly one of those categories.

The following is the debug version, slow but records details about overlap among categories:

```
/* check classification's postcondition:
 *  any of the categories above may be empty, but every label-goto pair
 *  should belong to exactly one of the categories above
 */
for( (lab, apps) <- gL.allPairs; app <- apps ) {
       val cats = Array.fill(10)(false)

       cats(0) = gL.whileLoops.contains(lab)
       cats(1) = gL.doWhileLoops.contains(lab)

       cats(2) = siColl.glSiblings exists { s => (s.lab == lab) && (s.app == app) }
       cats(3) = siColl.lgSiblings exists { s => (s.lab == lab) && (s.app == app) }

       cats(4) = dirColl.lgDirRelsDiffStmt exists { dr => (dr.lab == lab) && (dr.app == app) }
       cats(5) = dirColl.glDirRelsDiffStmt exists { dr => (dr.lab == lab) && (dr.app == app) }
       cats(6) = dirColl.directRelsSameStmt exists { p => (p.lab == lab) && (p.app == app) }

       cats(7) = indiColl.areInDiffStmts(app, lab)
       cats(8) = indiColl.areInDiffIfBranches(app, lab)
       cats(9) = indiColl.areInDiffSwitchCases(app, lab)

   val catCount: Int = cats.filter(c => c).size

   if(catCount != 1) { error("classification postcondition does not hold") }
}
```

4.6 Gaining details about tree shapes

Many of the `label-goto` pairs in a Scala AST are not loops. This subsection and the next show examples of those tree shapes.

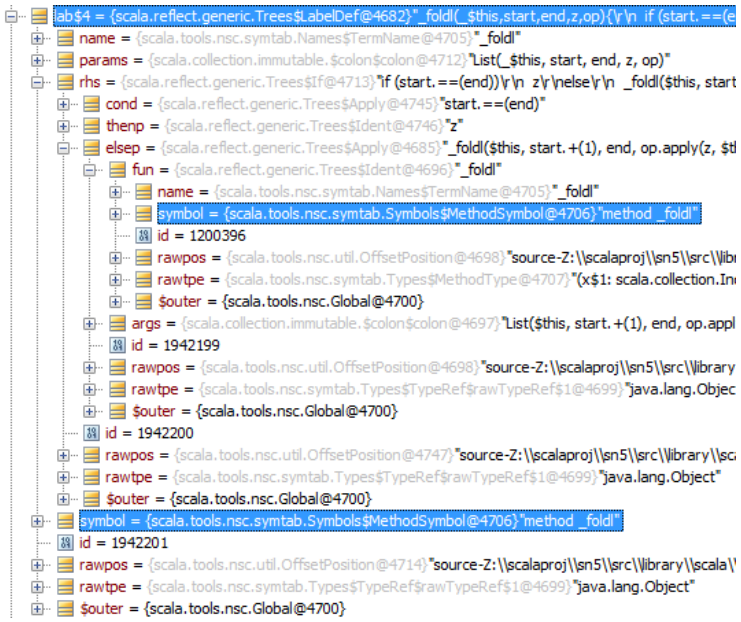


Figure 3: Sec. 4.6

One way to visualize `goto-label` pairs involves the debugger's views. For example, the pair in Figure 3 differs from a while-loop in that the `goto` is in the `else` branch (of the `if` expression contained in the `LabelDef`'s `rhs`). Additionally, the other branch does not contain `()` but `Ident(z)`.

Cases where `goto` and `label` are far away can also be visualized, with the help of `-uniqid` and `nodeToString`:

```
if(catCount != 1) {
  error("classification postcondition does not hold for\n\tlab symbol : \n" +lab.symbol+ "\n\tapp symbol : \n"
  val str = nodeToString(unit.body)
  scala.Console.println(str)
}
```

But for the above to provide useful information one needs to add the following to `NodePrinters.scala`:

```
case lab @ LabelDef(name, params, rhs) =>
  println("LabelDef(" + nodeinfo(tree))
  println(" \"" + name + "\",")
  for (param <- params)
    traverse(param, level + 1, true)
  traverse(rhs, level + 1, false)
  printcln(")")
```

Now that we are talking about tree shapes, I should mention a difference in the way `gotoelim` classifies `goto-label` pairs as compared to [2].

If we followed to the letter the formulation in that paper, we should 'unnest' from a `LabelDef` with `Block` as `rhs`, the stmts and expr there, so that any jump into the (immediately) 'containing' `LabelDef` is counted as a sibling-pair.

Instead, following the spirit of a `LabelDef` as a container of sub-expressions,

we classify such configuration as a directly-related pair (of the "`directRelsSameStmt`" variety). Other examples (with a different shape) of "directly-related pairs of the `directRelsSameStmt` variety" are:

```
_foldl#816432(_$this#816440,start#74452,end#74453,z#74454,op#74455){
  if (start.==#5143(end#74453))
    z#74454
  else
    _foldl#816432($this#1220817, start.+#5155(1), end#74453, op.apply#37879(z#74454, $this.apply#29146(start#744
}
```

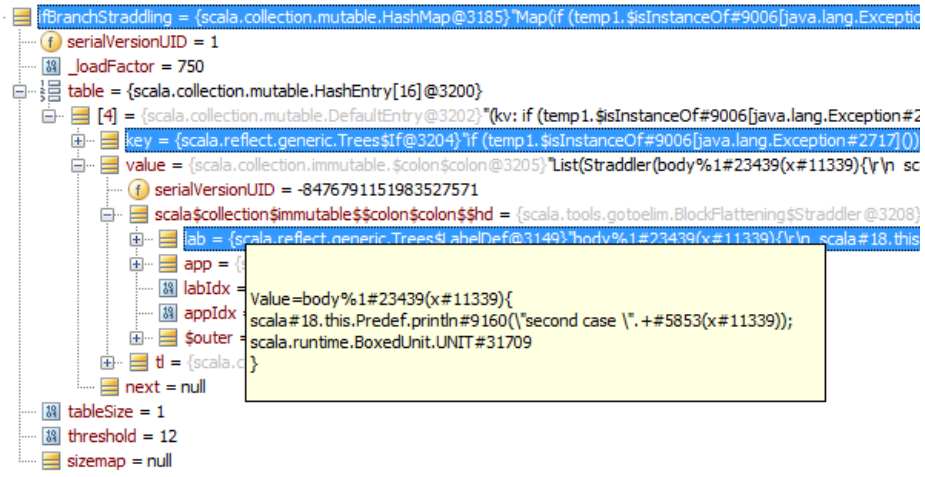
```
_loop#817037(_$this#817040,left#83862){
  if (left.>#5151(0).&&#4354($this.hasNext#34120()))
    {
      $this.next#34121();
      _loop#817037($this#1231632, left.-#5157(1))
    }
  else
    $this#1231632
}
```

In case you want to find more, just place a breakpoint as shown below at `DirectRelCollector.traverse()`:

```
469           case lab: LabelDef =>
470             val appsWithin = for(app <- gL.allGotosIn(lab); if !areLoopEndPoints(lab, app) )
471                              yield app
472             for (app <- appsWithin; if(isGotoLabelPair(app, lab))) {
473               // label at i, goto inside i
474               directRelsSameStmt = GLPair(lab, app) :: directRelsSameStmt
475             }
476           case _ => ()
477         }
478         super.traverse(tree)
```

4.7 In terms of our running example

The "`YouBadPatternMatcher`" example from Sec. 2.1 (shown in Listing 1 on p. 18) contains an indirectly-related pair, which straddles the branches of an `if` statement, as shown below:



5 Adapting the Erosa-Hendren algorithm to Scala

5.1 Transform for sibling pairs

```
TODO
```

5.2 Outward movement transform

Applies to:

- those directly related pairs (collected in `DirectRelCollector.directRels`) where the `goto` has a deeper level than the `label`.
- all "indirectly related" pairs (`entirelyDiffStmts`, `ifBranchStraddling`, `switchCaseStraddling`).

Sub-cases: out of a loop, out of a `switch`, out of an `if/else`.

```
TODO
```

5.3 Inward movement transform

Applies to those directly related pairs (collected in `DirectRelCollector.directRels`) where the `label` has a deeper level than the `goto`.

```
TODO
```

5.4 Putting it all together

Algorithm in Sec. 2.5 of [2].

```
TODO
```

6 Related Work

6.1 Decompiling Scala into Java

There have been attempts at decompiling classfiles emitted by `scalac`, into Java 1.4. For most Scala programs, no decompiler can in general recover compile-again, semantics-preserving Java sources. GOTO elimination should improve that situation. Additionally, one can give the decompiler an easier time with `-Ystruct-dispatch:no-cache -no-specialization`. Before `gotoelim`, the only option to have "less GOTOs" was `-Yno-squeeze -Ypmat-naive`.

- James Hamilton, Sebastian Danici.
  An Empirical Evaluation of Java Bytecode Decompiler Effectiveness[4]
- Dava decompiler[5]

6.2 Translating Scala ASTs into JavaScript

- ScalaGWT
  - `http://code.google.com/p/scalagwt`
  - `http://twitter.com/scalagwt`
- S2JS (Scala to Closure-annotated Javascript),
  `http://groups.google.com/group/s2js`

7 Future Work

7.1 A new backend for the Scala.Net compiler to emit C# sources

- `http://lamp.epfl.ch/~magarcia/ScalaNET/2011Q1/CSharpGen.pdf`

The conversion put forward above is related but different from another one (obtaining 3-address code from ICode) also described at *The Scala Compiler Corner* (for an early C#-emitting prototype). In the meantime, it became clear that `GenCSharp` can replace the `GenICode` → `GenMSIL` compilation pipepline, which has the advantage that the required transformations, once developed, can be reused across a larger number of plugins (including but not limited to those consuming ICode.)

References

[1] Iulian Dragos. *Compiling Scala for Performance*. PhD thesis, Lausanne, 2010. `http://lamp.epfl.ch/~dragos/files/dragos-thesis.pdf`.

[2] Ana M. Erosa and Laurie J. Hendren. Taming Control Flow: A Structured Approach to Eliminating GOTO Statements. In *ICCL*, 1994.

[4] `http://jameshamilton.eu/sites/default/files/JavaBytecodeDecompilerSurveyExtended.pdf`

[5] `http://www.sable.mcgill.ca/dava/`

Listing 1: Sec. 2.1

```
[[syntax trees at end of cleanup]]// Scala source: bt4.scala
package <empty>#3 {
  class YouBadPatternMatcher#9362 extends java.lang.Object#2344 with ScalaObject#430 {
    def problematicPattern#9367(): java.lang.Object#2344 = {
      var exceptionResult1#31723: java.lang.Object#2344 = _;
      try {
        exceptionResult1#31723 = scala.Int.box#4385(0)
      } catch {
        case (ex$1#11810 @ _) => {
          exceptionResult1#31723 = {
            {
              /*- start of block B */
              <synthetic> val temp1#20068: java.lang.Throwable#2266 = ex$1#11810;
              if (temp1.$isInstanceOf#9003[java.lang.Exception#2716]())
                {
                  /*- start of block B.T */
                  <synthetic> val temp2#20069: java.lang.Exception#2716 = temp1.$asInstanceOf#9005[java.lang.Exce
                  val x#11234: java.lang.Exception#2716 = temp2#20069;
                  if (YouBadPatternMatcher#9362.this.gd1$1#11822(x#11234))
                    {
                      /*- start of block B.T.T */
                      {
                        scala#21.this.Predef.println#9157("first case ".+#5852(temp2#20069));
                        scala.runtime.BoxedUnit.UNIT#31704
                      }
                    }
                  else
                    {
                      /*- start of block B.T.F */
                      val x#11336: java.lang.Throwable#2266 = temp2#20069;
                      body%1#23436(x#11336) { /*- this is a LabelDef tree node, the target of a jump. */
                        scala#21.this.Predef.println#9157("second case ".+#5852(x#11336));
                        scala.runtime.BoxedUnit.UNIT#31704
                      }
                    }
                }
              else
                {
                  /*- start of block B.F */
                  body%1#23436(temp1#20068); /*- this is an Apply node,
                       denoting a jump from an instruction in block B.F into an instruction inside B.T.F */
                  scala.runtime.BoxedUnit.UNIT#31704
                }
              /*- end of block B */
            }
          }
        }
      };
      exceptionResult1#31723
    };

    final <synthetic> private[this] def gd1$1#11822(x$1#11823: java.lang.Exception#2716): Boolean#3772 =
      x$1.getMessage#11299().==#5841("test");

    def this#9366(): YouBadPatternMatcher#9362 = {
      YouBadPatternMatcher#9362.super.this#5823();
      ()
    }
  }
}
```

Listing 2: Sec. 3.3

```
case Match(selector, cases) =>
  if (settings.debug.value)
    log("Generating SWITCH statement.");
  var ctx1 = genLoad(selector, ctx, INT)
  val afterCtx = ctx1.newBlock
  var caseCtx: Context = null
  generatedType = toTypeKind(tree.tpe)

  var targets: List[BasicBlock] = Nil
  var tags: List[Int] = Nil
  var default: BasicBlock = afterCtx.bb

  for (caze @ CaseDef(pat, guard, body) <- cases) {
    assert(guard == EmptyTree)
    val tmpCtx = ctx1.newBlock
    pat match {
      case Literal(value) =>
        tags = value.intValue :: tags
        targets = tmpCtx.bb :: targets
      case Ident(nme.WILDCARD) =>
        default = tmpCtx.bb
      case _ =>
        abort("Invalid case statement in switch-like pattern match: " +
              tree + " at: " + (tree.pos))
    }

    caseCtx = genLoad(body, tmpCtx, generatedType)
    caseCtx.bb.closeWith(JUMP(afterCtx.bb) setPos caze.pos)
  }
  ctx1.bb.emitOnly(
    SWITCH(tags.reverse map (x => List(x)), (default :: targets).reverse) setPos tree.pos
  )
  afterCtx
```

Listing 3: Sec. 2.2

```
-----------------------------------------------
original
-----------------------------------------------

if (evalB) {
  if (evalBT) {
    BTT
  } else {
    BTF
  }
} else {
  goto lBTF
}


-----------------------------------------------
step 1: goto out of if
-----------------------------------------------

var takeLBTF = false /* added */
if (evalB) {
  if (evalBT) {
    BTT
  } else {
    takeLBTF = false /* added */
    BTF
  }
} else {
  takeLBTF = true    /* replaced */
}
gotoif(takeLBTF, lBTF) /* added */


-----------------------------------------------
step 2: goto lifting (because goto appears after target)
-----------------------------------------------

var takeLBTF = false
do {                   /* added */
  gotoif(takeLBFT, lBTF) /* added */
  if (evalB) {
    if (evalBT) {
      BTT
    } else {
      takeLBTF = false
      BTF
    }
  } else {
    takeLBTF = true
  }
} while (takeLBFT)     /* changed */
```

Listing 4: Sec. 2.2

```
-----------------------------------------------
step 3: inward movement (goto into if, then branch)
-----------------------------------------------

var takeLBTF = false
do {
                          /* gotoif deleted */
  if (takeLBFT || evalB) { /* pre-prended OR-rand */
    gotoif(takeLBFT, lBTF) /* added */
    if (evalBT) {
      BTT
    } else {
      takeLBTF = false
      BTF
    }
  } else {
    takeLBTF = true
  }
} while (takeLBFT)


-----------------------------------------------
step 4: inward movement (goto into if, else branch)
-----------------------------------------------

var takeLBTF = false
do {
  if (takeLBFT || evalB) {
                                    /* gotoif deleted */
    if ( (!takeLBFT) && evalBT) {  /* pre-prended negated AND-rand */
      BTT
    } else {
      gotoif(takeLBFT, lBTF)       /* added */
      takeLBTF = false
      BTF
    }
  } else {
    takeLBTF = true
  }
} while (takeLBFT)


-----------------------------------------------
step 5: goto elim
-----------------------------------------------

var takeLBTF = false
do {
  if (takeLBFT || evalB) {
    if ( (!takeLBFT) && evalBT) {
      BTT
    } else {
                                  /* gotoif deleted */
      takeLBTF = false
      BTF
    }
  } else {
    takeLBTF = true
  }
} while (takeLBFT)
```