InlineExceptionHandlersPhase

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

November 24th, 2011

Abstract

"Inlining an exception handler H of a `try` T" consists in replacing a `THROW(clazz)` ICode instruction in T's body with a `JUMP` to a spliced-in `BasicBlock` that duplicates the entry block of H (provided it has been determined that H invariably catches exceptions thrown at runtime by the `THROW(clazz)` in question, Sec. 2). This rewriting preserves semantics because:

1. The spliced-in `BasicBlock` is not protected by any handler in T (instead, it's protected by all handlers that protect T itself);
2. Regarding basic-block-successors:
   (a) Before splicing, T has just one successor S, which is also the successor of H (ignoring `RETURN`s).
   (b) After splicing, the `THROW` has been replaced with a `JUMP` to the spliced-in `BasicBlock`, which keeps its original successor (ignoring `RETURN`s, that successor is S).

For brevity, we talk about a "duplicate handler" although in fact only its entry block is duplicated. This duplication occurs once per handler, i.e., all original `THROW` instructions whose `catch` handler can be no other than H will share a single H duplicate (Sec. 3). The rewriting does not change the `IMethod`'s `exh` list of `ExceptionHandler` other than by making the spliced-in `BasicBlock` be `covered` in the same way as T's body is.

```
          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
                   . . .
             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
/*----------------------------------------------------------------------*/
inlineExceptionHandlers 22 optimization: inline exception handlers
/*----------------------------------------------------------------------*/
            closelim 23  optimization: eliminate uncalled closures
                 dce 24  optimization: eliminate dead code
                 jvm 25  generate JVM bytecode
            terminal 26  The last phase in the compiler chain
```

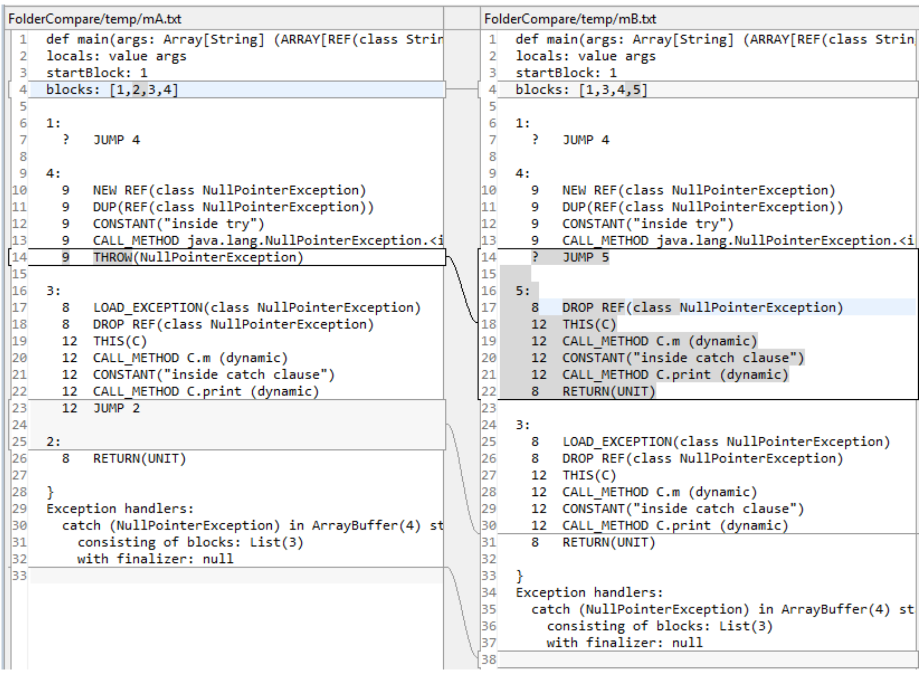

Figure 1: Sec. 1

Contents

1 Overview

Say we have:

```
try {
  throw new NullPointerException("inside try")
} catch {
  case _: NullPointerException =>
    m.print("inside catch clause")
}
```

After the `inlineExceptionHandlers` phase has run we have (Figure 1 on p. 2).

After splicing-in a `BasicBlock`, in general a new TFA (type-flow analysis) should be computed because:

```
  // notify the successors changed for the current block
  // notify the predecessors changed for the inlined handler block
  bblock.touched = true
  newHandler.touched = true
```

TFAs are expensive. Instead, we can postpone computing a new one and continue iterating *those basic-blocks that haven't been spliced-in* (their entry type-stacks had better not changed due to the splicing). This technique is demonstrated by the snippet below (`todoBlocks` contains those basic-blocks added as part of `applyBasicBlock`):

```
private def applyMethod(method: IMethod): Unit = {
  if (method.code ne null) {
    // create the list of starting blocks
    todoBlocks = global.icodes.linearizer.linearize(method)

    while (todoBlocks.nonEmpty) {
      val levelBlocks = todoBlocks
      todoBlocks = Nil
      levelBlocks foreach applyBasicBlock // new blocks will be added to todoBlocks
    }
  }

  . . .
```

Another way to avoid computing `MethodTFA`s is shown in Sec. 3:

> *this block was not analyzed, but it's a copy of some other block so its type-stack should be the same*

2 Finding a handler `H` none of whose predecessors can catch the exception type of interest (that `H` catches, "`findExceptionHandler()`")

Say a given `THROW(clazz)` is protected by a handler `H2` that catches `clazz`, but a previous catch-clause `H1` accepts a subtype of `clazz`. We can't be sure that the `THROW` in question will always lead to `H2`. Thus in this case it's not inlined. Example (quoting from the source comments):

Listing 1: Sec. 2

```
def findExceptionHandler(
  thrownException: TypeKind,
  handlersStarts: List[BasicBlock]
): Option[(BasicBlock, TypeKind)] = {

  for (handler <- handlers ; LOAD_EXCEPTION(clazz) <- handler take 1) {
    val caughtException = toTypeKind(clazz.tpe)

    // we'll do inlining here: createdException <:< thrownException <:< caughtException, good!
    if (thrownException <:< caughtException)
      return Some((handler, caughtException))

    // we can't do inlining here, the handling mechanism is more precise than we can reason about
    if (caughtException <:< thrownException)
      return None

    // no result yet
  }

  None
}
```

```
try {
  val exception: Throwable =
    if (cond) new IllegalArgumentException("even")
    else     new StackOverflowError("odd")
  throw exception
} catch {
  case e: IllegalArgumentException => . . . // H1
  case e: StackOverflowError =>   . . .
  case t: Throwable =>            . . . // H2
}
```

Rather than inspecting the `ExceptionHandler`'s `cls` field,

```
class ExceptionHandler(val method: IMethod, val label: String, val cls: Symbol, val pos: Position)
```

we can also find out which exception-type is caught based on the stack-manipulating ICode instruction `LOAD_EXCEPTION(clazz)`. To recap,

```
/** Fake instruction. It designates the VM who pushes an exception
 *  on top of the /empty/ stack at the beginning of each exception handler.
 *  Note: Unlike other instructions, it consumes all elements on the stack!
 *        then pushes one exception instance.
 */
case class LOAD_EXCEPTION(clasz: Symbol) extends Instruction {
  override def consumed = sys.error("LOAD_EXCEPTION cleans the whole stack")
  override def produced = 1
  override def producedTypes = REFERENCE(clasz) :: Nil
}
```

Upon iterating over catch-clauses, as long as the caught-type and the static-thrown-type are not comparable the outcome is inconclusive (no inlining in case all caught-types are non-comparable). The first comparable pair *(static-thrown-type, caught-type)* leads to making a decision on inlining (Listing 1).

3 Grabbing a handler duplicate (“duplicateExceptionHandlerCache(

The idea is to avoid duplicating the same handler twice, and knowing when we've tried and failed. That's the purpose of the `handlerCopies` map:

```
/* This map is used to keep track of duplicated exception handlers
 * explanation: for each exception handler basic block, there is a copy of it
 *
 * - some exception handler basic blocks might not be duplicated because
 *   they have an unknown format, that's why "Option[(...)]"
 *
 * - some exception handler duplicates expect the exception on the stack
 *   while others expect it in a local, that's why "Option[Local]"
 */
private val handlerCopies =
  perRunCaches.newMap[BasicBlock, Option[(Option[Local], BasicBlock)]]
```

Provided the start-block of the catch-handler in question has the “expected format” (i.e., it starts with `LOAD_EXCEPTION(caughtClass)`) a `copy` duplicate is obtained, some blocks are touched, and `copy` is added to the waiting list for :

```
  handlerCopiesInverted(copy) = ((handler, caughtException))
  todoBlocks ::= copy
```

The `handlerCopiesInverted` map allows cutting on the number of type-flows analyses performed (to just one!) per method. The following phrase cues about that trick:

> *this block was not analyzed, but it's a copy of some other block so its stack should be the same*

In detail:

```
  tfaCache.getOrElse(bblock.label, {
    // this block was not analyzed, but it's a copy of some other block so its stack should be the same

    val (origBlock, exception) = handlerCopiesInverted(bblock)

    val typeInfo            = getTypesAtBlockEntry(origBlock)

    val stack               =
      if (handlerLocal(origBlock).nonEmpty) Nil // empty stack, the handler copy expects an empty stack
      else List(exception)                  // one slot on the stack for the exception

    // If we use the mutability property, it crashes the analysis
    tfa.lattice.IState(new analysis.VarBinding(typeInfo.vars), new icodes.TypeStack(stack))
  })
```

4 Replacing a THROW instruction (“applyBasicBlock()”)

Assuming that a unique handler has been determined (Sec. 2) and its entry `BasicBlock` could be duplicated (Sec. 3) the remaining re-wiring is done in `applyBasicBlock()`. Basically, “a `THROW` is replaced with a `JUMP`” but that's only part of the story. There are two points of variation:

1. right before the THROW, the operand stack may or may not contain other values besides the exception on top; and

2. the handler may expect its exception in a local variable or on the stack. The former is indicated by exceptionLocalOpt being a Some(local) and the latter by None.

The four combinations above are covered by the following three case clauses:

```
// Prepare the new code to replace the THROW instruction
val newCode: List[Instruction] = exceptionLocalOpt match {

  /*- the handler expects its exception in local var */
  case Some(local) =>
    STORE_LOCAL(local) +: typeInfo.tail.map(x => DROP(x)) :+ JUMP(newHandler)

  /*- the handler expects its exception on the stack,
      which contains just the exception. */
  case None if typeInfo.length == 1 =>
    JUMP(newHandler) :: Nil

  /*- the handler expects its exception on the stack,
      but there are other values on it besides the exception. */
  case _ =>
    val exceptionType = typeInfo.head
    val localName  = currentClass.cunit.freshTermName("exception$")
    val localType  = exceptionType
    val localSymbol = bblock.method.symbol.newValue(NoPosition, localName).setInfo(localType.toType)
    val local      = new Local(localSymbol, localType, false)
    bblock.method.addLocal(local)
    STORE_LOCAL(local) :: typeInfo.tail.map(x => DROP(x)) ::: List(LOAD_LOCAL(local), JUMP(newHandler))

}
```

- *Nota bene*: Storing an exception in a local was chosen in a way that avoids the JVM-bytecode verification error (due to the interplay with "defined values analysis") described on p. 25 of the report: http://www.sable.mcgill.ca/publications/techreports/sable-tr-2003-3.pdf (that report is a very good read as it covers optimization of exception handling code).

5 Duplicating the handler ("duplicateExceptionHandler()")

Whatever else an exception handler does, its ICode starts with the LOAD_EXCEPTION instruction (which has no counterpart in bytecode, it simulates for type-flow purposes the operation of the VM placing the thrown exception on the stack). The handler duplicate ("copy"), being plain code, can't contain that instruction (nor is needed) thus it's not included in the duplicate. Please notice that saving the exception to a variable will be done before jumping to the duplicate (Sec. 4) and thus a STORE_LOCAL(local) instruction isn't copied either:

```
handler take 2 match {
  case Seq(LOAD_EXCEPTION(caughtClass), next) =>
    val (dropCount, exceptionLocal) = next match {
      case STORE_LOCAL(local) => (2, Some(local)) // we drop both LOAD_EXCEPTION and STORE_LOCAL
```

```
      case _                => (1, None)      // we only drop the LOAD_EXCEPTION and expect the exception on th
    }

    val caughtException = toTypeKind(caughtClass.tpe)

    val copy = handler.code.newBlock
    copy.emitOnly(handler drop dropCount: _*)
  . . .
```

A few highlights:

1. The duplicate is made to be protected by those handlers protecting the original handler (i.e., protecting the `Try` as a whole)

   ```
   // extend the handlers of the handler to the copy
   for (parentHandler <- handler.method.exh ; if parentHandler covers handler) {
     parentHandler.addCoveredBlock(copy)
     // notify the parent handler that the successors changed
     parentHandler.startBlock.touched = true
   }
   ```

2. Something that isn't changed: the successor of the duplicate, which results from the `JUMP` or `RETURN` appearing as last instruction in the duplicate.

3. `duplicateExceptionHandler()` can receive as argument only a `BasicBlock` that is the `startBlock` of an `ExceptionHandler`. This is a consequence of:

   ```
   (handler, caughtException) <- findExceptionHandler(toTypeKind(clazz.tpe), bblock.exceptionSuccessors)
   ```

   In case the catch-clause consists of a CFG with more than one `BasicBlock`, only the entry block is duplicated.

6 How does this optimization work in Scala.Net?

If left as-is, it doesn't work, because CIL VMs disallow jumps from the outside into a `catch`-block (as would happen in a multi-block exception handler inlined by `inlineExceptionHandlers`). What about the single-block case? There's the issue of leaving an exception handler according to CIL rules, as discussed next.

Bottom line: looks like `inlineExceptionHandlers` should join this club in `JavaPlatform`:

```
def platformPhases = List(
  flatten,  // get rid of nested classes
  genJVM    // generate .class files
) ++ depAnalysisPhase
```

- **A note on terminology**: In this subsection we adopt the terminology of the CIL standard and employ"`try`-block" to mean a CIL *range of instructions* (in general, a Scala-level `try`-expression results in a number of ICode `BasicBlock`s which are finally mapped to a range of consecutive CIL instructions). Same goes for `catch`-block (i.e., a range of instructions derived from a single `catch`-clause in a Scala `catch` partial function) and for `finally`-block.

Structured Exception Handling (SEH) on the Microsoft CLI imposes more stringent requirements than its JVM counterpart (details in [1]). Regarding entering `catch`-blocks, in essence the spec states:

- Control flow arrives to a `catch`-block (resp. `finally`-block) only when transfered by the execution system, thus ruling out fall-through and jumps (from outside) as means to enter into those blocks (Partition I, §12.4.2.8.1, "*Entry to filters or handlers can only be accomplished through the CLI exception system*").

Regarding leaving `catch`-blocks (details in [1]):

- The instruction `ret` shall not be enclosed in a protected block, or handler (where "handler" encompasses `catch`-blocks and `finally`-blocks).
- a jump instruction enclosed in a `try`-block must remain intra-block (resp. `catch`-block, resp. `finally`-block)
- the CIL `throw` instruction may appear freely in a `try`-block, `catch`-block, or `finally`-block. Same goes for the CIL `rethrow` instructions, but ICode does not have such instruction.
- no `try`-block, `catch`-block, or `finally`-block may be left by fall-through. In these cases, the MSIL backend emits `leave <blockAfterTry>` to leave a `catch`-clause and `endfinally` to leave a `finally`-block.

References

[1] Miguel Garcia. Exception handling: from ICode to CIL, 2010. Notes at *The Scala Compiler Corner*. `http://lamp.epfl.ch/~magarcia/ScalaCompilerCornerReloaded/2010Q2/ExceptionHandling.pdf`.