Exception handling: from ICode to CIL

© Miguel Garcia, LAMP,
École Polytechnique Fédérale de Lausanne (EPFL)
http://lamp.epfl.ch/~magarcia

May 9th, 2010

Contents

Abstract

Emitting .NET assemblies requires following certain rules (static semantics) for code participating in Structured Exception Handling (SEH). `GenMSIL` includes support (`MSILlinearizer`) for this task, relying on `ilasm` syntax to demarcate protected and handler blocks. Assembly-emitting APIs (for example, CCI) expect the incoming CIL to already fulfill well-formedness rules (WFRs) for SEH. These notes describe the underlying exception handling model, control-flow aspects of ICode relevant to these WFRs, and how to generate correct CIL in the presence of exception handling when emitting assemblies directly.

1 Static semantics of Exception Handling in CIL

In this section all the rules making up the Structured Exception Handling (SEH) model of CIL are reviewed. 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.

1.1 Entering a `try`-block

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 reduced to a `finally` protecting a `try-catch` expression, and this approach is followed by ICode.

As will be seen later, ICode uses an instance of `ExceptionHandler` (Sec. 2) for each of the following (in a single `try-catch-finally` occurrence):

- each clause in the `catch` partial function;
- the `finally` block.

The CIL standard spells out (Partition I, §12.4.2) the allowable control flow patterns for `try-catch-finally`:

- Control flow can arrive to a `try`-block either by fall-through from the previous instruction or by a jump to the very first instruction of the block (but not by "jump into the middle" of it).
- A jump targeting an instruction (other than the first) within a `try`-block can only originate in another instruction within the same `try`-block. Two notes:
  - the same restriction holds for jump instructions enclosed in `catch`-blocks and in `finally`-blocks: source and target of a jump in one such block must be internal to that very same block.
  - a CIL `jmp` instruction denotes "jump to method start discarding the current stack frame". What ICode calls "jump" corresponds to a `br` (branch) CIL instruction. In running text, we'll use the term "jump" in the sense of ICode jump.

Moreover (quoting from §12.4.2.8.1):

> *Entry to protected blocks can be accomplished by fall-through, at which time the evaluation stack shall be empty.*

There are CIL code idioms that simplify obeying these rules. For example, a `try`-block with more than one `BasicBlock` as predecessors can be entered by fall-through from a `nop` instruction inserted just to be the new jump target. For the purposes of exception handling, the protected range of instructions starts with the instruction lexically following that `nop`. Intra-`try` jumps don't have to be adjusted to account for the just inserted `nop`. In particular, an intra-`try` jump-to-block-start has to remain as-is, in order for the target (the start of the protected range of instructions) to remain intra-`try`.

1.2 Entering a `catch`-block or a `finally`-block

In essence, the spec says:

- 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.

Unlike for a `try`-block, it's not possible to enter into them by jumping (from outside) to the start of the block. The above is based on Partition I, §12.4.2.8.1:

> *Entry to filters or handlers can only be accomplished through the CLI exception system; that is, it is not valid for control to fall through into such blocks. This means filters and handlers cannot appear at the beginning of a method, or immediately following any instruction that can cause control flow to fall through.*
>
> *Conditional branches can have multiple effects on control flow. Since one of the possible effects is to allow control flow to fall through, a filter or handler cannot appear immediately following a conditional branch.*

The CIL code idioms that keep us out of trouble (with respect to the above) are the same as those required to leave such blocks (the topic of Sec. 1.3). Due to the nature of `catch`-blocks and `finally`-blocks, no dedicated CIL instructions are to be emitted to effect entering into them, as this task is performed by the execution system alone.

1.3 Leaving a `try`-block, `catch`-block, or `finally`-block

The rules imposed by the CLR in this regard are:

- The instruction `ret` shall not be enclosed in a protected block, or handler (where "handler" encompasses `catch`-blocks and `finally`-blocks).
- the aforementioned "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.

Although no `rethrow` will be emitted from ICode, you may be curious to know its well-formedness rules anyway:

> *Correct CIL uses this instruction only within the body of a catch handler (not of any exception handlers embedded within that catch handler).*
>
> `rethrow` *does not consume an exception value from the stack (nor requires it to be there, i.e. it performs neither "pop" nor "peek").*

1.3.1 Leaving a `finally`-block

Leaving a `finally`-block is easy: the only two alternatives are `throw` and `endfinally`. The latter takes no argument:

- for normal control flow, the execution system will not fall-through. Instead, the current `finally`-block was entered either upon running into a `throw` (or `rethrow`) instruction, or because of a `leave` *`<target>`* instruction. In both cases, the execution of the `finally`-block gets sandwiched in-between by the execution system, and control is transferred after `endfinally` to the next layer of the sandwich (based on the originating `throw` or `leave` *`<target>`*). More on this in Sec. 1.3.4.
- for exceptional control flow, the execution system will transfer the current exception as appropriate.

For example, in case more than one ICode-level `finally`-blocks have the same `BasicBlock` as successor (if this can't happen in ICode assume an unconstrained CFG instead), it is then necessary to insert jump instructions immediately after each `finally`-block (resp. `leave` instructions if enclosed themselves in a `try`-block, `catch`-block; or `endfinally` if enclosed in a `finally`-block).

In CIL, `ret` can't appear in a `finally`-block:

> *12.4.2.8.2.3 `ret`:*
>
> *Shall not be enclosed in any protected block, filter, or handler.*
>
> *Note: To return from a protected block, filtered handler, or catch handler, a `leave(.s)` instruction is needed to transfer control to an address outside all exception-handling blocks, then a `ret` instruction is needed at that address. end note*
>
> *Note: Since the `tail.` prefix on an instruction requires that that instruction be followed by `ret`, tail calls are not allowed from within protected blocks, filters, or handlers. end note*

In Scala, the type of a `finally`-block has to be `scala.Nothing`. Another CIL rule:

> *An `endfinally` instruction can only appear lexically within a `finally` block. Unlike the `endfilter` instruction, there is no requirement that the block end with an `endfinally` instruction, and there can be as many `endfinally` instructions within the block as required.*

1.3.2 Leaving a `try`-block

Leaving any of a `try`-block or `catch`-block again offers only two alternatives: `throw` or `leave` *`<target>`*. A `leave` instruction has to be used to indicate the normal control flow, thus avoiding fall-through as a means to exit the block in question. However it may come as a bit of surprise that the *`<target>`* when leaving a `try`-block is *not* the `finally`-block (only the execution system may take us there ...) but whatever comes after the accompanying `catch` or `finally` block:

- in the case of a `try`-block, `leave` should target an instruction after the accompanying `catch`-block (for a `try-catch`) or after the accompanying `finally`-block (for a `try-finally`). This target instruction may have been inserted just for this purpose. For example, the C# compiler (for a debug build) exits the innermost `try` in

  ```
  try { try {} catch {} } finally {}
  ```

  by leaving first to a `nop` followed by another `leave`, where both of these two instructions appear immediately after and outside the `catch`-block. The target of this second `leave` lies immediately after (thus outside) the `finally`-block.

1.3.3 Leaving a `catch`-block

For the reasons sketched above,

- in the case of a `catch`-block, `leave` should target an instruction immediately after the `catch`-block itself (no `finally` comes after a `catch` in the desugared form, only `try-catch` and `try-finally` exist). In keeping with the same example, the C# compiler (for a debug build) exits the `catch` in

  ```
  try { try {} catch {} } finally {}
  ```

  by leaving to the same instructions injected as targets of the preceding `try` (as discussed in the previous subsection). Those instructions are a `nop` followed by another `leave`.

1.3.4 Sidenote: there's no fall-through from a `finally`-block

Using `ildasm` we re-arrange a block to occur immediately after the last instruction being covered by a `finally`-block, recompile with `ilasm`, and successfully `peverify` the result (Listing 2).

The CIL in question was compiled from:

```
public static void Main(String[] args)
{
    if (System.DateTime.Now.Millisecond == 0)
    {
        try { System.Console.WriteLine("then branch, in try"); }
        catch (Exception) { System.Console.WriteLine("then branch, in catch"); }
        finally { System.Console.WriteLine("then branch, in finally"); }
    }
    else
    {   try { System.Console.WriteLine("else branch, in try"); }
        catch (Exception) { System.Console.WriteLine("else branch, in catch"); }
        finally { System.Console.WriteLine("else branch, in finally"); }
    }
}
```

and produces the following output (unless milliseconds == 0):

```
else branch, in try
else branch, in finally
afterEndFinally
myNewInstructions
```

If fall-through occurred, the string "myNewInstructions" would appear on the console after "else branch, in finally". Instead, the exception entry

```
.try IL_004e to IL_0070 finally handler IL_0070 to IL_007e
```

declares that exiting the range from `IL_004e until IL_0070` should result in running the finally block (`handler IL_0070 to IL_007e`). Therefore, the `endfinally` on line `IL_007d` instructs the execution system to take over, which completes execution of `leave.s   IL_007f`.

1.4 Some rules about stack-emptiness

> *`endfinally` empties the evaluation stack as a side-effect*
>
> *The evaluation stack shall be empty after the return value is popped by a `ret` instruction.*
>
> *Exception handlers can access the local variables and the local memory pool of the routine that catches the exception, but any intermediate results on the evaluation stack at the time the exception was thrown are lost.*
>
> *An exception object describing the exception is automatically created by the CLI and pushed onto the evaluation stack as the first item upon entry of a `filter` or `catch` clause.*

As a corollary from the last rule, no user-accessible exception is available on the evaluation stack when entering a CIL `finally`-block. If throwing an exception from there, push it beforehand!

2 ICode idioms for Exception Handling

2.1 `ExceptionHandler`

ICode uses an instance of `ExceptionHandler` (Sec. 2) for each of the following (in a single `try-catch-finally` occurrence):

- each clause in the `catch` partial function;
- the `finally` block.

The constructor of `ExceptionHandler` reads:

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

An `IMethod` keeps a field:

```
/** The list of exception handlers, ordered from innermost to outermost. */
var exh: List[ExceptionHandler] = Nil
```

Based on that list, an array of exception entries (for a CIL method) will be populated. When talking about these entries, it makes sense to mentally replace “code block” with “range of instructions”, given that a reference to a block will be encoded as an `(offset, length)` pair.

Coming back to `ExceptionHandler`, besides the constructor args, it has fields: (a) `resultKind: TypeKind` (for the MSIL backend), (b) `startBlock: BasicBlock`, (c) `covered: Set[BasicBlock]`, (d) `blocks: List[BasicBlock]`.

The invariants of an `ExceptionHandler e` are:

- `e.cls` is
  - the `tpe.typeSymbol` of the exception caught (a subtype of `definitions.ThrowableClass`) for a `catch`-clause in a `catch` partial function
  - `NoSymbol` for a `finally`-block
- `e.covered`, a `Set[BasicBlock]`, representing the protected instructions
- `e.blocks`, a `List[BasicBlock]`, representing the handler instructions (of which `e.startBlock` is their entry point)
- `e.blocks contains e.startBlock`

TODO ok, not all of the above are "invariants", and in fact the interesting ones are those when the `IMethod` is known (like, whether mutual protection is allowed, which regions must be either disjoint or one completely nested in the other, etc.)

A `BasicBlock` b may be preceded by a single `BasicBlock` a whose only successor is b. These two blocks are not collapsed into one: one but not the other belongs to `exh.covered` or `exh.blocks` for some `exh`.

2.2 Sample ICode for `synchronized` under JVM

When calling `Context.Try`, the correct version (JVM or MSIL) is chosen behind the scenes. However the invoker (`ICodePhase.genSynchronized` in our example) gets to decide the instructions that go into each `catch`-clause in a `catch`-block, and what `Tree` to translate as `finally`-block. These invokers are platform-independent.

As warm-up, we'll look in this section at the `BasicBlocks` and `ExceptionHandlers` generated for JVM from this example:

```
def main(args: Array[String]) {
   synchronized {
      scala.Console.println("in synchro")
   }
}
```

To make a long story short, the expression e_1 `synchronized` e_2 gets translated into four basic-blocks (Listing 1), that we call `enterE1`, `evalE2`, `bbRet`, and `bbCatch`.

- The first block (`enterE1`) pushes e_1's value onto the stack, dups, saves it in a local variable (for `monitorexit` later), enters the monitor, and transfers control unconditionally to `evalE2`.
- Block `evalE2` evaluates e_2, saves the result (if any) in a local variable (because the stack is cleared just before entering `catch`-blocks or `finally`-blocks), and transfers control to `bbRet`.

Listing 1: `HelloSynchronized`

```
def main(args: Array[java.lang.String] (ARRAY[REFERENCE(java.lang.String)])): Unit {
locals: value args, variable monitor1
startBlock: 1
blocks: [1,2,3,4]

1:            // enterE1
  26  THIS
      DUP
      STORE_LOCAL variable monitor1
  26  MONITOR_ENTER
  26  JUMP 4

4:            // evalE2
  27  LOAD_MODULE object Console
  27  CONSTANT (Constant(in synchro))
  27  CALL_METHOD scala.Console scala.Console.println (dynamic)
  27  LOAD_FIELD scala.runtime.BoxedUnit.UNIT
  27  DROP REFERENCE(scala.runtime.BoxedUnit)
      LOAD_LOCAL variable monitor1
  26  MONITOR_EXIT
  26  JUMP 2

2:            // bbRet
  26  RETURN (UNIT)

3:            // bbCatch
      LOAD_LOCAL variable monitor1
  26  MONITOR_EXIT
      THROW

}
Exception handlers:                    //   def printExceptionHandler(e: ExceptionHandler) {
  catch (Throwable) in Set(4) starting at: 3 // e.cls.simpleName , e.covered , e.startBlock
    consisting of blocks: List(3)     // e.blocks
    with finalizer: null              // e.finalizer
    // not shown: linearizer.linearize(e.startBlock) foreach printBlock;
```

- `bbRet` pushes the result (`UNIT` if none) and returns.
- `bbCatch` contains a `monitorexit` (that's also the last thing `evalE2` does), as well as a `THROW`. Taken together, these instructions leave the stack empty at the end of the `catch`-block. We'll see later that the other invoker of `Context.Try` (`genLoadTry`) manipulates the stack differently in a `catch`-clause. Each such clause leaves on the stack whatever the `catch`-clause computes, but before doing that it removes the exception value from the stack (either with `DROP(REFERENCE(sym))` or with `STORE_LOCAL(exception)`).

The final result of ICode generation looks as shown in Listing 1.

Can the ICode in Listing 1 be translated into CIL? (a hypothetical question because the for-JVM version of `Context.Try` was used). e_1 `synchronized` e_2 has in effect been desugared as if coming from the following pseudo-Java (or pseudo-CIL, for that matter):

```
ReturnTypeIfAny res = zero of ReturnType
AnyRef mon = e1
```

```
try {
  Lock(mon)
  res = e2
  Unlock(mon)
} catch (Throwable tmp) { /*- see note Throwable vs.System.Exception below */
  Unlock(mon)
  throw tmp
}
return res
```

The JVM-style catch-all `catch (Throwable tmp)` is expressed in CIL as `catch (System.Object tmp)`, because non-CLS compliant languages (e.g., C++) can throw an arbitrary object as exception. The idiom `catch(Exception e)` that pervades C# 3.0 code works as catch-all because [1, §8.10]:

> *In the current implementation of C# and the CLR, by default a thrown object that does not derive from `Exception` is converted to a `RuntimeWrappedException` object. As a consequence, `catch (Exception e)` catches all exceptions. If you want to disable this behavior and use the C# 1.0 semantics ...*

Assuming we were to translate into CIL the ICode in Listing 1, block `bbCatch` should ("obviously") be declared as a CIL `catch`-block (not as a `finally`-block) otherwise the `THROW` instruction would be rejected by `peverify`, because when entering a CIL `finally`-block (irrespective of exceptional or normal control flow) there's no (CIL-accessible) exception on the stack.

2.3 Comparison with the C# compiler

Before seeing what the for-MSIL version of `Context.Try` generates, let's see how the C# compiler translates the `lock` construct:

```
public void lockingTest()
{
    Object thisLock = new Object();
    lock (thisLock) { /* Critical code section */ }
}
```

The CIL generated by the C# compiler is:

```
.method public hidebysig instance void lockingTest() cil managed
{
  // Code size     42 (0x2a)
  .maxstack 2
  .locals init ([0] object thisLock,
          [1] bool '<>s__LockTaken0',
          [2] object CS$2$0000,
          [3] bool CS$4$0001)
  IL_0000: nop
  IL_0001: newobj    instance void [mscorlib]System.Object::.ctor()
  IL_0006: stloc.0
  IL_0007: ldc.i4.0
  IL_0008: stloc.1
  .try
  {
    IL_0009: ldloc.0
    IL_000a: dup
```

```
    IL_000b: stloc.2
    IL_000c: ldloca.s  '<>s__LockTaken0'
    IL_000e: call      void [mscorlib]System.Threading.Monitor::Enter(object,
                                                                    bool&)
    IL_0013: nop
    IL_0014: nop
    IL_0015: nop
    IL_0016: leave.s   IL_0028
  }  // end .try
  finally
  {
    IL_0018: ldloc.1
    IL_0019: ldc.i4.0
    IL_001a: ceq
    IL_001c: stloc.3
    IL_001d: ldloc.3
    IL_001e: brtrue.s  IL_0027
    IL_0020: ldloc.2
    IL_0021: call      void [mscorlib]System.Threading.Monitor::Exit(object)
    IL_0026: nop
    IL_0027: endfinally
  }  // end handler
  IL_0028: nop
  IL_0029: ret
} // end of method WhatAboutExHs::lockingTest
```

Not that it's terribly related or anything, but I'll mention this anyway: `http://channel9.msdn.com/posts/bruceky/Whirlwind-14-Whats-new-in-C-4-Events/`

> *This session on C# 4 shows how the compiler handles eventing. The compiler no longer sets locks on events, instead uses a compare and swap technique. Stuart compares C# from .NET 3.5 and .NET 4 to show the differences and explains the implications for your existing code.*

3 Codegen aspects of `try-catch-finally` common to JVM and CIL

Unlike what happens inside `Context.Try`, the code preparing arguments for it is platform-independent. The arguments thus prepared determine the ICode instructions going into `BasicBlocks` for each `catch`-clause (and for the `finally`-block if any).

Because control-flow instructions face restrictions under CIL's Structured Exception Handling (SEH) model (Sec. 1), we want to know which instructions of that kind may appear in ICode produced by `Context.Try`, i.e. in the `BasicBlocks` referred from an `ExceptionHandler`'s `covered` and `blocks` fields.

For example, we saw (Sec. 2.2 and Sec. 3.1) that if

e_1 `synchronized` e_2

returns a value, the generated ICode `RETURN` instruction will be outside of the protected block and the handler block. This already fulfills one of the requirements placed by the CIL SEH model. To emit correct CIL, it only remains emitting the appropriate CIL `leave` instructions branching to that `RETURN` (instead of the ICode `JUMPs` that are in place).

The arguments to `Context.Try` are prepared as described below (the source comment is longer):

```
/**
 * ctx.Try( ctx => {
 *                  ctx.bb.emit(...) // protected block
 *               },
 *   List(
 *    (ThrowableClass,           // case clause 1 -> ExceptionHandler
 *     resTypeOfTryAsTypeKind,
 *     ctx => { ctx.bb.emit(...);
 *           }
 *     ),
 *    (AnotherExceptionClass,    // case clause 2 -> ExceptionHandler
 *     resTypeOfTryAsTypeKind,
 *     ctx => { ...
 *           }
 *     )
 *   ),
 *   finalizer, //  -> ExceptionHandler
 *   tryTree
 * )
 */
def Try(body: Context => Context,
        handlers: List[(Symbol, TypeKind, (Context => Context))],
        finalizer: Tree,
        tree: Tree) = if (forMSIL) TryMsil(body, handlers, finalizer, tree) else {
```

The above how-to is useful when preparing arguments to invoke `Context.Try`, as done in exactly two places in the compiler:

- `ICodePhase.genSynchronized` (Sec. 2.2), and
- `ICodePhase.genLoadTry`.

Please notice that `finalizer` (the third component of a `Try`)

- is `EmptyTree` whenever `Context.Try` is invoked from `genSynchronized`
- is passed as-is from `genLoadTry` to `Context.Try`.

```
private def genLoadTry(tree: Try, ctx: Context, setGeneratedType: TypeKind => Unit): Context = {
  val Try(block, catches, finalizer) = tree
  val kind = toTypeKind(tree.tpe)

  val caseHandlers =
    for (CaseDef(pat, _, body) <- catches.reverse) yield {
      def genWildcardHandler(sym: Symbol): (Symbol, TypeKind, Context => Context) =
        (sym, kind, ctx => {
          ctx.bb.emit(DROP(REFERENCE(sym)))
          genLoad(body, ctx, kind)
        })

      pat match {
        case Typed(Ident(nme.WILDCARD), tpt) => genWildcardHandler(tpt.tpe.typeSymbol)
        case Ident(nme.WILDCARD)          => genWildcardHandler(ThrowableClass)
        case Bind(name, _)                =>
          val exception = ctx.method addLocal new Local(pat.symbol, toTypeKind(pat.symbol.tpe), false)

          (pat.symbol.tpe.typeSymbol, kind, {
```

```
          ctx: Context =>
            ctx.bb.emit(STORE_LOCAL(exception), pat.pos);
            genLoad(body, ctx, kind);
        })
      }
    }

  ctx.Try(
    bodyCtx => {
      setGeneratedType(kind)
      genLoad(block, bodyCtx, kind)
    },
    caseHandlers,
    finalizer,
    tree)
}
```

BTW, `genLoadTry` is the first line of defense when `GenICode` visits a `Try` tree:

```
case t @ Try(_, _, _) => genLoadTry(t, ctx, (x: TypeKind) => generatedType = x)
```

3.1 ICode shapes produced by `genSynchronized`

The details about `Context.Try` just mentioned allow stating more precisely the shapes of ICode trees that `ICodePhase.genSynchronized` produces:

1. no need to worry about the `finalizer` argument passed to `Context.Try` by `ICodePhase.genSynchronized` because it's `EmptyTree`. The instructions for what intuitively constitutes a `finally`-block (exiting the monitor) appear both:
   - in the block `exh.blocks` (`exh` is the single `ExceptionHandler` generated from e_1 `synchronized` e_2), and
   - near the end of (the only block in the list) `exh.covered`, after evaluating e_2 (the first argument to `synchronized`).
2. "no need to worry" means that, if translated into CIL verbatim, the `THROW` instruction (for re-throwing) does not occur in a CIL `finally`-block but in a `catch`-block, where an exception value is always available on the stack (in a CIL `finally`-block instead we would have to take care to load it ourselves).
3. also in `exh.covered`, the result (assuming there's one) of evaluating e_2 does not remain on the stack:

   ```
   ctx1 = ctx1.Try(
     bodyCtx => {
       val ctx2 = genLoad(args.head, bodyCtx, expectedType /* toTypeKind(tree.tpe.resultType) */)
           if (hasResult)
             ctx2.bb.emit(STORE_LOCAL(monitorResult))
   ```

4. that's only part of the story, i.e. the arguments passed to `Context.Try`. Further ICode instructions may be added after returning from that invocation, and `genSynchronized` adds some. Well, not many but just one:

```
if (hasResult)
 ctx1.bb.emit(LOAD_LOCAL(monitorResult))
```

5. coming back to "preparing arguments for `Context.Try`", the argument for the only `catch`-clause includes one instruction to re-throw the exception sitting on the stack. This instruction will go to the single block referred from `exh.covered`.

The invocations from `genSynchronized` of `Context.enterSynchronized(monitor: Local)` and `Context.exitSynchronized(monitor: Local)` do not emit any instructions but update the `Context`-local `cleanups` field:

```
def enterSynchronized(monitor: Local): this.type = {
  cleanups = MonitorRelease(monitor) :: cleanups
  this
}

def exitSynchronized(monitor: Local): this.type = {
  assert(cleanups.head == monitor, "Bad nesting of cleanup operations: " +
         cleanups + " trying to exit from monitor: " + monitor)
  cleanups = cleanups.tail
  this
}
```

TODO Do the JVM and MSIL versions of `Context.Try` differ in case there's no `finally`-block? (that's the case produced by `genSynchronized`)

3.2 ICode shapes produced by genLoadTry

The `Tree` received by `genLoadTry` has components:

```
Try(block: Tree, catches: List[CaseDef], finalizer: Tree)
```

- `finalizer` is passed as-is to `Context.Try`, i.e. so far no decision is made on the resulting ICode instructions.
- `block` (i.e., the `try`-block) is also passed as-is to `Context.Try`.
- each `catch`-clause is a `CaseDef(pat, _, body)`. The resulting `startBlock` in the corresponding `ExceptionHandler` will pop the exception, either with:
  - a `DROP(REFERENCE(sym))`, in case no variable was specified (i.e., `pat` is `Typed(Ident(nme.WILDCARD), tpt)` or `Ident(nme.WILDCARD)`), or
  - a `STORE_LOCAL(exception)`, where `exception` stands for a local variable created based on `pat` of the form `Bind(name, _)`.

  In both cases the stack will be empty by the time the ICode instructions emitted by the ensuing `genLoad(body, ctx, kind);` are executed.

For reference, here goes the code listing:

```
private def genLoadTry(tree: Try, ctx: Context, setGeneratedType: TypeKind => Unit): Context = {
  val Try(block, catches, finalizer) = tree
  val kind = toTypeKind(tree.tpe)
```

```
  val caseHandlers =
    for (CaseDef(pat, _, body) <- catches.reverse) yield {
      def genWildcardHandler(sym: Symbol): (Symbol, TypeKind, Context => Context) =
        (sym, kind, ctx => {
          ctx.bb.emit(DROP(REFERENCE(sym)))
          genLoad(body, ctx, kind)
        })

      pat match {
        case Typed(Ident(nme.WILDCARD), tpt)  => genWildcardHandler(tpt.tpe.typeSymbol)
        case Ident(nme.WILDCARD)              => genWildcardHandler(ThrowableClass)
        case Bind(name, _)                    =>
          val exception = ctx.method addLocal new Local(pat.symbol, toTypeKind(pat.symbol.tpe), false)

          (pat.symbol.tpe.typeSymbol, kind, {
            ctx: Context =>
              ctx.bb.emit(STORE_LOCAL(exception), pat.pos);
              genLoad(body, ctx, kind);
          })
      }
    }

  ctx.Try(
    bodyCtx => {
      setGeneratedType(kind)
      genLoad(block, bodyCtx, kind)
    },
    caseHandlers,
    finalizer,
    tree)
}
```

4 Control-flow instructions generated by `Context.TryMsil`

4.1 For the `finally`-block

```
if (finalizer != EmptyTree) {
  val exh = outerCtx.newHandler(NoSymbol, UNIT)
  this.addActiveHandler(exh)
  // exh covers try and all catches, i.e. try { try {..} catch {..} } finally {..}
  val ctx = finalizerCtx.enterHandler(exh)

  /*- as discussed in Sec. 1.4, stack will be empty on entry to CIL finally.
      Semantics of LOAD_EXCEPTION discussed below.
      Added for Xdce purposes, should not be emitted by GenCIL */
  if (settings.Xdce.value) ctx.bb.emit(LOAD_EXCEPTION())

  /*- expectedType == UNIT (left on top of the stack) */
  val ctx1 = genLoad(finalizer, ctx, UNIT)

  /*- every ICode BasicBlock needs a control-transfer instruction (JUMP here) at the end */
  ctx1.bb.emit(JUMP(afterCtx.bb))
  ctx1.bb.close
  finalizerCtx.endHandler()
}
```

After running this, `exh.startBlock` is `ctx.bb`, and `exh.blocks` comprises (in addition to `exh.startBlock`) any blocks produced by `genLoad(finalizer, ctx, UNIT)` (e.g. `ctx1.bb`). The resulting `exh.blocks` stretch from `LOAD_EXCEPTION` as first instruction to `JUMP(afterCtx.bb)` as last one.
And the semantics of `LOAD_EXCEPTION`:

```
/** Fake instruction. It denotes the VM pushing 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() extends Instruction {
  override def toString(): String = "LOAD_EXCEPTION"
  override def consumed = error("LOAD_EXCEPTION does clean the whole stack, no idea how many things it consumes!
  override def produced = 1
  override def producedTypes = AnyRefReference :: Nil
}
```

4.2 For each catch-clause

```
for (handler <- handlers) {
  val exh = this.newHandler(handler._1, handler._2) // newHandler(exceptionClassSymbol, expectedTypeOfTry)
  var ctx1 = outerCtx.enterHandler(exh)
  if (settings.Xdce.value) ctx1.bb.emit(LOAD_EXCEPTION()) /*- don't emit this in CIL */
  ctx1 = handler._3(ctx1)
  // msil backend will emit `leave` to jump out of a handler
  ctx1.bb.emit(JUMP(afterCtx.bb)) /*- 'leave' instead of 'JUMP' */
  ctx1.bb.close
  outerCtx.endHandler()
}
```

Similar to Sec. 4.1, the resulting `exh.blocks` stretch from `LOAD_EXCEPTION` as first instruction (in `exh.startBlock eq ctx1.bb`) to `JUMP(afterCtx.bb)` as last one.

4.3 For try-block, and two closing JUMPs

```
val outerCtx = this.dup     // context for generating exception handlers, covered by finalizer
val finalizerCtx = this.dup // context for generating finalizer handler
val afterCtx = outerCtx.newBlock

if (finalizer != EmptyTree) { ... // OMITTED, see Sec. 4.1 above
for (handler <- handlers) { ... // OMITTED, see Sec. 4.2 above

val bodyCtx = this.newBlock
val finalCtx = body(bodyCtx) /*- Note below */

/*- from the start of the outermost "try {}" into the start of the innermost "try {}" */
outerCtx.bb.emit(JUMP(bodyCtx.bb))
outerCtx.bb.close

// msil backend will emit `leave` to jump out of a try-block
/*- from the end of the (outermost) finally the BasicBlock after the "try {} finally {}" */
finalCtx.bb.emit(JUMP(afterCtx.bb))
finalCtx.bb.close
```

The `body: Context => Context` parameter of `TryMsil` receives as argument either the following (from `genLoadTry`):

```
bodyCtx => {
  setGeneratedType(kind)
  genLoad(block, bodyCtx, kind)
}
```

or the following (from `genSynchronized`)

```
bodyCtx => {
  val ctx2 = genLoad(args.head, bodyCtx, expectedType /* toTypeKind(tree.tpe.resultType) */)
  if (hasResult)
    ctx2.bb.emit(STORE_LOCAL(monitorResult))
  ctx2.bb.emit(Seq(
    LOAD_LOCAL(monitor),
    MONITOR_EXIT() setPos tree.pos
  ))
  ctx2
}
```

In both of these two cases, the ICode instructions emitted by `body(bodyCtx)` are ok with respect to CIL stack rules for protected regions (and contain only intra-range jumps if any).

5 Recovering `try-catch-finally` block nesting from `IMethod.exh`

This is performed by `MSILLinearizer.linearize(m: IMethod): List[BasicBlock]`. The intuition is:

1. partition all `ExceptionHandler`s (for the `IMethod` in question) having the same `exh.covered`. Each partition contains one or more `catch` clauses, or alternatively a `finally` block.

2. The `covered` sets of `BasicBlock`s for `exh`s in different partitions are either disjoint or one is a strict subset of the other.

Recovering block nesting is necessary:

- in `GenMSIL`, where the block-syntax for `try-catch` and `try-finally` is emitted (the less-readable but more flexible alternative is the "Label Form of EH Clause Declaration" [2, Ch. 14]):

  ```
  .try <label> to <label> <EH_type_specific> handler <label> to <label>

  <EH_type_specific> ::= catch <class_ref> | filter <label> | finally | fault
  ```

- when implementing a C# backend for the Scala.Net compiler[1]

TODO Run dead code elimination for the MSIL backend, to get rid of unreachable catch and finally clauses (for example, for an empty `try`-expression.)

[1] `http://lamp.epfl.ch/~magarcia/ScalaCompilerCornerReloaded/threeaddress.pdf`

TODO Does dead code elimination require `inliners` to perform its job? (`inliners` in turn requires reading bytecode into ICode, right?)

TODO I gues for the particular case of empty `try`, `catch`, or `finally` blocks a custom `dce` can be inserted before `MSILLinearizer`.

A comparison of the two syntaxes supported by `ilasm` for exception handling [2, Ch. 14]:

> *The only quality the label form lacks is convenience. In view of that, ILAsm offers an alternative form of EH clause description: a scope form ... The scope form can be used only for a limited subset of all possible EH clause configurations: the handler blocks must immediately follow the previous handler block or the guarded block ... The IL disassembler by default outputs the EH clauses in the scope format least those clauses that can be represented in this form. However, we have the option to suppress the scope form and output all EH clauses in their label form (command-line option `/RAW`).*

6 TODO: `NonLocalReturnControl`

```
package scala.util.control

/**
 * A marker trait indicating that the Throwable it is mixed into
 * is intended for flow control.
 *
 * Note that Throwable subclasses which extend this trait
 * may extend any other Throwable subclass (eg. RuntimeException)
 * and are not required to extend Throwable directly.
 *
 * Instances of Throwable subclasses marked in
 * this way should not normally be caught. Where catch-all behaviour is
 * required ControlThrowables should be propagated. For example:
 *
 *  import scala.util.control.ControlThrowable
 *
 *  try {
 *    // Body might throw arbitrarily
 * } catch {
 *   case ce : ControlThrowable => throw ce // propagate
 *   case t : Exception => log(t)        // log and suppress
 *
 * @author Miles Sabin
 */
trait ControlThrowable extends Throwable with NoStackTrace
```

```
package scala.runtime

import scala.util.control.ControlThrowable

class NonLocalReturnControl[T](val key: AnyRef, val value: T) extends ControlThrowable
```

References

[1] Anders Hejlsberg, Mads Torgersen, Scott Wiltamuth, and Peter Golde. *The C# programming language, third edition.* Addison Wesley Professional, 2008.

[2] Serge Lidin. *Expert .NET 2.0 IL Assembler.* Apress, Berkely, CA, USA, 2006.

Listing 2: No fall-through from a `finally`-block, CIL excerpt

```
   IL_004d: nop
   IL_004e: nop
   IL_004f: ldstr     "else branch, in try"
   IL_0054: call      void [mscorlib]System.Console::WriteLine(string)
   IL_0059: nop
   IL_005a: nop
   IL_005b: leave.s   IL_006d

   IL_005d: pop
   IL_005e: nop
   IL_005f: ldstr     "else branch, in catch"
   IL_0064: call      void [mscorlib]System.Console::WriteLine(string)
   IL_0069: nop
   IL_006a: nop
   IL_006b: leave.s   IL_006d

   IL_006d: nop
   IL_006e: leave.s   IL_007f

   IL_0070: nop
   IL_0071: ldstr     "else branch, in finally"
   IL_0076: call      void [mscorlib]System.Console::WriteLine(string)
   IL_007b: nop
   IL_007c: nop
   IL_007d: endfinally
   IL_007e: nop

myNewInstructions: ldstr    "myNewInstructions"
                  call      void [mscorlib]System.Console::WriteLine(string)
                  br IL_0081

   IL_007f: nop

afterEndFinally: ldstr    "afterEndFinally"
                call      void [mscorlib]System.Console::WriteLine(string)

   IL_0080: br myNewInstructions
   IL_0081: ret
   // Exception count 4
   .try IL_0019 to IL_0028 catch [mscorlib]System.Exception handler IL_0028 to IL_0038
   .try IL_0019 to IL_003b finally handler IL_003b to IL_0049
   .try IL_004e to IL_005d catch [mscorlib]System.Exception handler IL_005d to IL_006d
   .try IL_004e to IL_0070 finally handler IL_0070 to IL_007e
  } // end of method WhatAboutExHs::Main
```