ICode inlining

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

February 20th, 2012

Abstract

The inlining analysis [1, §3.3.1] inspects certain callsites at compile time, checking whether the pair *(static-type-of-receiver, callee-signature)* uniquely determines a concrete method (*the* concrete method that will be dispatched at runtime). If so, further checks are performed (regarding e.g. resulting code size) before proceeding to insert the callee's ICode instructions at the callsite. These notes describe implementation aspects of this optimization.

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

Contents

1 Overview

The `inliner` phase [1, §3.3.1] iterates over all `IClass`-es being compiled, skipping some methods (constructors, abstract methods, bridge methods, and those annotated with `@inline`) whose bodies won't be inspected:

```
def analyzeClass(cls: IClass): Unit = {
  if (settings.inline.value) {
    this.currentIClazz = cls
    for (imethod <- cls.methods;
         if !imethod.symbol.isConstructor;
         if  imethod.code != null;
         if !hasInline(imethod.symbol);
         if !imethod.symbol.isBridge) {
      analyzeMethod(imethod)
    }
  }
}
```

In turn, `analyzeMethod()` goes over the `BasicBlocks` of "*the caller*" looking for inlining opportunities in the form of `CALL_METHOD` instructions (which may target methods hosted in a library or being compiled in the same compiler run). Upon inlining, the current basic block is abandoned and iteration continues with the next basic block. Provided inlining occurred, the caller's body itself may be iterated more than once (`retry` variable), each time based on a new *type-flow analysis* providing a type-stack at the start of each basic block (afterwards, `tfa.interpret(currTypeStack, instr)` gives an updated post-instruction type-stack). Why a new type-flow analysis? For one, inlining adds at least two new basic blocks (Sec. 3). Additionally, more precise types might be computed with the inlined instructions in place.

```
  do {
    retry = false
    tfa init m
    tfa.run
    for(bb <- caller.linearized) {
      info = tfa in bb
      var bbUpdated = false
      for (i <- bb; if !bbUpdated) {
        i match {
          // Dynamic == normal invocations
          // Static(true) == calls to private members
          case CALL_METHOD(msym, Dynamic | Static(true))
          if !msym.isConstructor && !hasNoInline(msym) =>
            bbUpdated = analyzeInc(msym, i, bb)
          case _ => ()
        }
        if(!bbUpdated) info = tfa.interpret(info, i);
      }
    }
  } while (retry && count < MAX_INLINE_RETRY)
```

Before returning, `analyzeMethod()` glues together basic blocks as per:

```
/** Merge together blocks that have a single successor which has a
 * single predecessor. Exception handlers are taken into account (they
 * might force to break a block of straight line code like that).
 * This method should be most effective after heavy inlining.
```

```
 */
def normalize(): Unit = if (this.code ne null) { . . .
```

2 What to do with this callsite? ("`analyzeInc()`")

There are two aspects to the operation of:

```
def analyzeInc(calleeSym: Symbol, callsite: CALL_METHOD, bb: BasicBlock): Boolean
```

First, deciding whether inlining should take place (Sec. 2) and if so, clearing and populating again the basic block's instructions (Sec. 3) where the `callsite` occurs. In the latter case, `true` is returned (and only then).

2.1 Lookup of the unique callee-symbol to dispatch ("`lookupImplFor()`")

A precondition for callee inlining is availability of ICode for it, i.e. availability of the concrete method body that would be dispatched at runtime. Therefore, such `IMethod` can be looked up provided that the pair *(static-type-of-receiver, callee-signature)* uniquely determines a concrete method. The necessary and sufficient condition for that is a disjunction:

```
receiverClazz.isEffectivelyFinal || calleeSym.isEffectivelyFinal
```

(where `receiverClazz` was provided by type-flow analysis). In the former case, the (final) `receiverClazz` may itself lack an implementation for `calleeSym`, thus requiring walking up the super-class hierarchy to find the one being inherited. The logic in charge of this lookup is contained in `lookupImplFor()`:

```
/** Look up implementation of method 'sym in 'clazz'.
 */
def lookupImplFor(sym: Symbol, clazz: Symbol): Symbol = {
  // TODO:
  // verify that clazz.superClass is equivalent here to
  //           clazz.tpe.parents(0).typeSymbol (.tpe vs .info)

  . . .
```

```
TODO OPEN: Methods defined in traits are not inlined,
         \url{https://issues.scala-lang.org/browse/SI-4767}
```

2.2 Additional requisites for external methods ("`shouldLoadImplFor()`")

After the steps in Sec. 2.2 *the* `concreteMethod`-symbol has been found that safely predicts the outcome of runtime method dispatch. However, it might point to a definition in an external library. In such cases, the guard in `shouldLoadImplFor()` has "veto power" over the inlining decision (to recap, no ICode available means no inlining).

In short, a callee "living in an external library" will be loaded (Sec. 4) only if:

- annotated with `@inline` (this is the only way for an external, user-defined method to be considered for inlining),
- otherwise it lives in `scala.Predef`, or its class lives in `scala.runtime`, or it is one of the few "monadic" methods (`foreach`, `filter`, `withFilter`, `map`, `flatMap`) and is also `final`.

```
/** Should method 'sym' being called in 'receiver' be loaded from disk? */
def shouldLoadImplFor(calleeSym: Symbol, receiverClazz: Symbol): Boolean = {
  def alwaysLoad  =
        (receiverClazz.enclosingPackage == RuntimePackage)
     || (receiverClazz == PredefModule.moduleClass)

  def loadCondition =
        calleeSym.isEffectivelyFinal
     && isMonadicMethod(calleeSym)
     && isHigherOrderMethod(calleeSym)

  hasInline(sym) || alwaysLoad || loadCondition
}

private def isMonadicMethod(sym: Symbol) = {
  val (origName, _, _) = nme.splitSpecializedName(sym.name)
  origName match {
    case nme.foreach | nme.filter | nme.withFilter | nme.map | nme.flatMap => true
    case _ => false
  }
}

private def isHigherOrderMethod(sym: Symbol) =
  sym.isMethod && atPhase(currentRun.erasurePhase.prev)(sym.info.paramTypes exists isFunctionType)
```

2.3 And now that ICode is available: more requisites ("`isCandidate`", "`isStampedForInlining`", "`isSafeToInline`")

Whether externally loaded or compiled in this run, the `IMethod`s for caller and callee are subject to further checks:

1. non-overridability of the callee, where "callee" was looked-up as per Sec. 2.2. Non-overridability is given by any of

   ```
   def isCandidate = (
       isClosureClass(receiverClazz) || concreteMethod.isEffectivelyFinal || receiverClazz.isEffectivelyFina
   )
   ```

2. caller and callee aren't one and the same, the scoring heuristics give green light, and the callee's instructions don't make inlining unsafe (this last condition, `isSafeToInline()`, is expanded below).

   ```
   def isStampedForInlining(stack: TypeStack) =
     !sameSymbols   &&
      inc.hasCode   && /*- ie. the callee's IMethod has non-null `code` field.*/
      shouldInline  &&
      isSafeToInline(stack)

   private def sameSymbols = caller.sym == inc.sym
   ```

```
private def neverInline = caller.isBridge || !inc.hasCode || inc.noinline
private def alwaysInline = inc.inline

/** Decide whether to inline or not. Heuristics:
 *   - it's bad to make the caller larger (> SMALL_METHOD_SIZE) if it was small
 *   - it's bad to inline large methods
 *   - it's good to inline higher order functions
 *   - it's good to inline closures functions.
 *   - it's bad (useless) to inline inside bridge methods
 */
def shouldInline: Boolean = !neverInline && (alwaysInline || {
  /*- scoring-based heuristics */
  . . .
}
```

The vetoing conditions that exclusively depend on the callee's instructions are encapsulated in `isSafeToInline()`:

```
/** A method is safe to inline when:
 *    - it does not contain calls to private methods when called from another class
 *    - it is not inlined into a position with non-empty stack,
 *      while having a top-level finalizer (see liftedTry problem)
 *    - it is not recursive
 * Note:
 *    - synthetic private members are made public in this pass.
 */
def isSafeToInline(stack: TypeStack): Boolean = {
  . . .
```

3 Inserting the callee's instructions ("`CallerCalleeInfo.doInline()`")

```
/** Inline 'inc' into 'caller' at the given block and instruction.
 */
def doInline(block: BasicBlock, instr: CALL_METHOD)
```

At this point, the callee goes by the name of "`inc`". Keeping in mind that it has a single entry point, the inlining mechanics are:

1. make room for the CFG of the callee, by leaving in the basic block containing the callsite ("`block`") only those instructions until the callsite.
2. splice a copy of the callee's CFG as a successor of the current block,
3. the instructions originally following the inlined callsite go into a new block ("`afterBlock`") which also becomes the successor of some spliced blocks after reformulating the callee's `RETURN` statements:

   ```
   case RETURN(_) => JUMP(afterBlock)
   ```

There are more moving parts, but the above already conveys the essentials.

4 Parsing the callee's instructions ("ICodeReader")

The unit of loading is an `IClass`. Once parsed from bytecode, they are tracked separately (in `icodes.loaded`) from those built by `GenICode` (in `icodes.classes`):

```
/** The icode of the given class, if available */
def icode(sym: Symbol): Option[IClass] = (classes get sym) orElse (loaded get sym)

/** Load bytecode for given symbol. */
def load(sym: Symbol) {
  try {
    val (c1, c2) = icodeReader.readClass(sym)

    assert(c1.symbol == sym || c2.symbol == sym,
      "c1.symbol = %s, c2.symbol = %s, sym = %s".format(c1.symbol, c2.symbol, sym))
    loaded += (c1.symbol -> c1)
    loaded += (c2.symbol -> c2)
  } catch {
    case e: Throwable => // possible exceptions are MissingRequirementError, IOException and TypeError -> no be
      log("Failed to load %s. [%s]".format(sym.fullName, e.getMessage))
      if (settings.debug.value)
        e.printStackTrace
  }
}
```

```
TODO Question:

  "The unit of loading is an IClass".
  Does this mean that failure to parse any single method
  (whether candidate for inlining or not, whether invoked from an inlining candidate or not)
  renders all methods in that IClass non-available for inlining?

TODO Gather stats on how often this happens.
```

Some highlights:

1. Exception entries aren't parsed, thus it wouldn't be possible, say, to decompile `try-catch-finally` from the parsed ICode:

```
val exceptionEntries = in.nextChar.toInt
var i = 0
while (i < exceptionEntries) {
  // skip start end PC
  in.skip(4)
  // read the handler PC
  code.jmpTargets += in.nextChar
  // skip the exception type
  in.skip(2)
  i += 1
}
skipAttributes()
```

2. After parsing all instructions, a *customized* type-flow analysis and a reaching-defs analysis may be needed (Sec. 4.1 and Sec. 4.2):

```
if (code.containsDUPX)
  code.resolveDups()
```

```
if (code.containsNEW)
  code.resolveNEWs()
```

4.1 LinearCode.resolveDups()

`GenJVM` does not emit them, but the stack-manipulation instructions `DUP_X1`, `DUP_X2`, `DUP2_X1`, and `DUP2_X2` may be found while parsing bytecode.

They are reformulated by `resolveDups()` into a sequence of equivalent instructions, using temporary locals instead.

The CIL instructions set deliberately avoids those instructions, as well as `swap` ("swaps two top words on the stack (note that value1 and value2 must not be double or long)".

- JVM instructions:
  `http://en.wikipedia.org/wiki/Java_bytecode_instruction_listings`
- Differences between JVM and CLR:
  `http://www.daimi.au.dk/~beta/ooli/Compare.html`

4.2 LinearCode.resolveNEWs()

In Scala ASTs, the invocation of a constructor, i.e.

```
case Apply(fun @ Select(New(tpt), nme.CONSTRUCTOR), args)
```

is lowered into a several ICode instructions (push reference to new object, duplicate it, load arguments, call instance initializer). Quoting from `GenICode`:

```
val nw = NEW(rt)
ctx.bb.emit(nw, tree.pos)
ctx.bb.emit(DUP(generatedType))
val ctx1 = genLoadArguments(args, ctor.info.paramTypes, ctx)

val init = CALL_METHOD(ctor, Static(true))
nw.init = init /*- this field will be needed by Inliner */
ctx1.bb.emit(init, tree.pos)
ctx1
```

When reading back ICode, `resolveNEWs()` tries to recognize which `CALL_METHOD` correspondes to each `NEW` instruction, otherwise `dumpMethodAndAbort()`.

In other words, ICode follows the JVM pattern for object-creation:

```
/** Creating objects works differently on .NET. On the JVM
 *  - NEW(type) => reference on Stack
 *  - DUP, load arguments, CALL_METHOD(constructor)
 *
 * On .NET, the NEW and DUP are ignored, because the NewObj opcode does their job instead.
 *  - load arguments
 *  - NewObj(constructor) => reference on stack
```

5 The New Inlining Algorithm (except that, it has a lot in common with the old one)

Just in time for Scala 2.10, the inlining algorithm was refactored[1], achieving a significant speedup. In order to understand the new algorithm, let's take stock of the information necessary and sufficient for inlining. To recap, basic blocks are iterated to find callsites whose *(static-type-of-receiver, callee-signature)* qualify as candidates for a deeper check via `analyzeInc()`. The new algorithm collects that information as a side-effect of type-flow analysis (TFA). This additional work is performed in a subclass of `MethodTFA` that is used only by `Inliner` (`MTFAGrowable` is the subclass in question).

The above by itself does not cut down on TFA effort. Before getting there, let's see what inlining does to the caller's CFG, which will be the key to avoiding computing afresh a full TFA. Instead, we will *repair* the existing solution.

- The mechanics of inlining (Sec. 3) modify in-place the caller's CFG by (a) trimming some instructions from the block where the callsite was hosted; (b) splicing in a number of new blocks; and (c) connecting what used to be exit instructions in the callee by jumps to the "`afterBlock`", another new block, that contains the instructions trimmed from the original basic block. All in all, a call instruction is replaced with a single-entry single-exit CFG that is embedded in the caller's CFG.
- In terms of an iterative dataflow analysis, nodes affected by (a) have a stale lattice element at block exit ("out-flow"); while nodes added by (b) and (c), being new, have no lattice elements whatsoever (neither on block entry nor block exit).

Given the division of labor between `Inliner` (in charge of updating in-place the caller's CFG) and `MTFAGrowable` (TFA computation) all that `doInline()` can do about (a), (b), and (c) above is conveying that information for the TFA to be repaired (this communication occurs by invoking `MTFAGrowable.reinit()`). With that information on hand, TFA repair should focus on blocks reachable from blocks having stale out-flows. This includes all new blocks, be they inlined or "`afterBlock`". That's why we add all blocks with stale out-flows to the TFA's worklist (only). In due course, lattice elements will be computed where needed (thus bringing up to date, or repairing, the TFA solution).

This brings us back to the TFA analysis. It does more than pushing and popping blocks to the working list, applying the transfer function in between. It does more in order to do less. You see, as the type of a receiver is lub-bed from that of block predecessors, we might notice the *(static-type-of-receiver, callee-signature)* does not qualify anymore as candidate for inlining. That means we can remove it from a "watchlist" (a set of `CALL_METHOD` instructions), that is useful in connection with another set (of basic blocks, "`relevantBBs`"). Useful because they allow us to quit applying the transfer function whenever the TFA has reached the perimeter of the CFG subgraph of interest. Details? There are some hefty source comments in `Inliners.scala` and `TypeFlowAnalysis.scala`.

[1] `https://github.com/scala/scala/commit/6255c482572441e729a59e448adfa12d338752bc`

Listing 1: Sec. A

```
*** Cumulative statistics at phase inliner
ms type-flow-analysis  : 140668
ms copy-propagation    :      0
ms liveness-analysis   :      0
ms reachingDefinitions :      0

*** Cumulative statistics at phase inlineExceptionHandlers
ms type-flow-analysis  : 147973
ms copy-propagation    :      0
ms liveness-analysis   :      0
ms reachingDefinitions :      0

*** Cumulative statistics at phase closelim
ms type-flow-analysis  : 147973
ms copy-propagation    :   2021
ms liveness-analysis   :    750
ms reachingDefinitions :    750


*** Cumulative statistics at phase dce
ms type-flow-analysis  : 147973
ms copy-propagation    :   2021
ms liveness-analysis   :   1738
ms reachingDefinitions :   1738
```

A Where does time go?

The measurement "`ms type-flow-analysis`" includes only the time spent in `MethodTFA.run()`. A more complete picture can be gained by including `MethodTFA.init()`. After adding other timers for the other dfa's (`CopyAnalysis`, `LivenessAnalysis`, `ReachingDefinitionsAnalysis`) we can see for the example of compiling the compiler (see also Listing 1):

```
[inliner in                 231708ms] // i.e. 68% of the compiler run
[inlineExceptionHandlers in  7753ms] //       2%
[closelim in                  4043ms] //       1%
[dce in                      17837ms] //       5%
. . .
[total in                   336324ms]
```

Focusing on the inliner phase, a useful distinction is between:

- External methods that are inlined in methods being compiled, Listing 3
- Methods being compiled that were inlined in methods being compiled, Listing 2

Parsing bytecode is actually not that expensive (as compared to computing type-flows). Moreover, about ten methods (Listing 3) account for the vast majority of this kind of inlining (in the example of compiling the compiler).

Regarding "'Methods being compiled that were inlined in methods being compiled" (Listing 2), the times `apply()` of an anon-closure were inlined is shown below. Any optimization here would help a lot.

Listing 2: Sec. A

```
Methods being compiled that were inlined in methods being compiled
times     (%) symbol
----- ------- ------
  214  (27.4%) scala.tools.nsc.Global.debuglog
  174  (22.3%) scala.tools.nsc.Global.log
  111  (14.2%) scala.reflect.internal.SymbolTable.atPhase
   43   (5.5%) scala.tools.nsc.interactive.Global.debugLog
   39   (5.0%) scala.reflect.internal.Symbols$Symbol.setFlag
   35   (4.5%) scala.reflect.internal.Symbols$Symbol.fullName
   22   (2.8%) scala.tools.nsc.interpreter.repldbg
   16   (2.0%) scala.reflect.internal.Symbols$Symbol.isOverloaded
   15   (1.9%) scala.tools.nsc.interactive.RefinedBuildManager$$anonfun$invalidated$2.scala$tools$nsc$interactiv
   14   (1.8%) scala.tools.nsc.ast.TreeDSL$CODE.mkTreeMethods
   14   (1.8%) scala.tools.nsc.typechecker.Namers$Namer$$anonfun$addDefaultGetters$2.scala$tools$nsc$typechecker
   14   (1.8%) scala.tools.nsc.typechecker.Typers$Typer.printInference
   13   (1.7%) scala.tools.nsc.typechecker.Namers$Namer$$anonfun$addDefaultGetters$2$$anonfun$apply$13.scala$too
   13   (1.7%) scala.tools.nsc.typechecker.Typers$Typer.printTyping
   12   (1.5%) scala.tools.nsc.backend.icode.GenICode.scala$tools$nsc$backend$icode$GenICode$$debugassert
   11   (1.4%) scala.tools.nsc.ast.TreeDSL$CODE.REF
   11   (1.4%) scala.tools.nsc.interactive.Global.informIDE
   10   (1.3%) scala.tools.nsc.ast.parser.Parsers$Parser.commaSeparated
 Other inlinings (fewer than ten times each method): 1489
 Times that getters/setters were inlined: 374
 Number of anon-closure's apply's that were inlined: 2584, of which 292 were $sp.
-------------------------------------------------
```

Number of anon-closure's apply's that were inlined: 2584, of which 292 were $sp.

B Suggestions to improve performance

B.1 Callee TFA (1 of 2): Caching of TFA for external methods

The TFA of an external callee doesn't change during compilation. But as of now, it will be re-computed as many times as that method is inlined. Better to cache it, right?

A similar argument applies to the TFA of the caller. If the most recent TFA is kept in a cache, there's no need to re-compute it (will be needed when that method plays the role of caller or callee in an inlining attempt).

Sidenotes:

1. After ICode has been loaded for the callee, a *customized* `MethodTFA` is initialized and run in some cases (as part of `resolveDups()`, Sec. 4.1). Afterwards it is discarded.

```
TODO
No need to discard it. The override of interpret() replaces DUPX opcodes
with functionally equivalent ICode STORE and LOAD instructions.
That does not change the type-stack at BasicBlock boundaries.
Anyway, DUPX are so infrequent that we won't gain much
by not discarding the MethodTFA instance.
```

Listing 3: Sec. A

```
External methods that were inlined in methods being compiled
times     (%) symbol
----- ------- ------
  264 (16.5%) scala.Predef$ArrowAssoc.$minus$greater
  258 (16.1%) scala.Predef.assert
  132  (8.2%) scala.Predef.augmentString
  128  (8.0%) scala.Option.getOrElse
   97  (6.0%) scala.Option.map
   83  (5.2%) scala.Predef.println
   83  (5.2%) scala.runtime.ScalaRunTime.inlinedEquals
   75  (4.7%) scala.LowPriorityImplicits.intWrapper
   68  (4.2%) scala.collection.immutable.Range.foreach$mVc$sp
   67  (4.2%) scala.Option.foreach
   63  (3.9%) scala.runtime.RichInt.until
   62  (3.9%) scala.collection.immutable.Range.apply
   43  (2.7%) scala.Option.flatMap
   37  (2.3%) scala.Predef.any2ArrowAssoc
   30  (1.9%) scala.Predef.any2stringadd
   27  (1.7%) scala.Predef.refArrayOps
   22  (1.4%) scala.Option.orElse
   19  (1.2%) scala.Option.exists
   16  (1.0%) scala.runtime.ScalaRunTime.hash
   15  (0.9%) scala.Option.filter
   15  (0.9%) scala.collection.immutable.Range.foreach
 Other inlinings (fewer than ten times each method): 64
 Times that getters/setters were inlined: 0
 Number of anon-closure's apply's that were inlined: 0, of which 0 were $sp.
-------------------------------------------------
```

The invocations

```
    if (code.containsDUPX)
      code.resolveDups()

    if (code.containsNEW)
      code.resolveNEWs()
```

are currently being done as part of `parseBytecode()` (see also Sec. 4.2). In case the method fails the `isSafeToInline(stack)` test, that work would have been in vain. Perhaps some `isSafeToInline(stack)` conditions can be evaluated before finishing polishing the parsed method.

```
TODO
Actually, resolveDups() and resolveNEWs() can be moved from LinearCode to IMethod
(their input is just the IMethod) so that they can be invoked only after
successful testing for inlining-safety.
Another place for those methods could be Inliner.
```

2. BTW, during `doInline()` the type-flow analysis of the callee is needed only to know how many `DROP` instructions to emit as part of replacing a `RETURN(UNIT)` or a `RETURN(kind)` (granted, except that `DROP` takes as argument the `TypeKind` of the stack top).

C Clarification needed

C.1 Questions

```
TODO Question about availability of ICode for receiverMethod,
    Why is that determined in two different ways?

 Way 1:
    def isAvailable = icodes available concreteMethod.enclClass
 i.e. the enclClass of the unique callee-symbol is deemed to hold the IMethod whose code will be inlined.

 Way 2:
   def lookupIMethod(meth: Symbol, receiver: Symbol): Option[IMethod] = {
     def tryParent(sym: Symbol) = icodes icode sym flatMap (_ lookupMethod meth)

     receiver.info.baseClasses.iterator map tryParent find (_.isDefined) flatten
   }
 the receiverClazz is used as starting point (which takes longer,
 and hopefully leads to the same result as Way 1).
```

C.2 Are these bugs?

```
TODO Looks like neither ICodeReader nor ClassfileParser are setting the parsed IMethod's exh field.
    Why do we have

       // add exception handlers of the callee
       caller addHandlers (inc.handlers map translateExh)

    in Inliners?
```

```
ICodeReader doesn't set IMethod.recursive after parseByteCode().
Therefore the vetoing condition about not being recursive in isSafeToInline()
applies in effect only to callees being compiled in this run.
Can this be a problem? Are externally-defined, recursive callees, vetoed in some other way?
```

```
In ICodeReader, the first index of a param list depends on whether the method is static or not:

var idx = if (method.isStatic) 0 else 1

ie. <MethodSymbol>.isStaticMember is queried.
However, in some cases (e.g. compiling LongMap) there is a discrepancy between that and
  ( (current_jflags & ch.epfl.lamp.fjbg.JAccessFlags.ACC_STATIC) != 0)

Perhaps related, javaToScalaFlags() doesn't inspect Java's ACC_STATIC flag.

Giving the wrong "first index" may later cause checkValidIndex() to fail.
```

C.3 How to learn more about the inliner

A query for `https://issues.scala-lang.org/secure/IssueNavigator!executeAdvanced.jspa` (for example, Figure 1).

```
project = SI
```

T	Key	Summary
	SI-4767	Methods defined in traits are not inlined
	SI-4579	infinite loop in the inliner
	SI-3569	"final var", public fields (!) and the inliner
	SI-5005	@specialize vs. @inline: @specialize wins (unnecessarily)
	SI-4925	Inner class methods not inlined when compiled separately

Figure 1: Sec. C.3

```
AND (summary ~ inlin OR description ~ inlin)
AND issuetype = Bug AND status = Open
```

References

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