Just trying to generate *faster* code *faster* under `-optimise`

Miguel Garcia
http://lamp.epfl.ch/~magarcia

LAMP, EPFL

2011-11-22

Outline

- Stats about inlining ("short-term")
  - Why focus on the inliner?
  - Inlining of "external" methods
  - Inlining of "internal" methods
  - Dealing with multiple inlinings of the same callee
- Early inlining of anonymous closures ("medium-term")
  - AST shapes of interest
  - Example 2: `Range.foreach`
  - Advantages
- Ideas for the future ("long-term")
  - Dedicated early-inlining to avoid `NonLocalReturn`s
  - Inlining using a stackless IR requires zero type-flow analyses
- Wrap-up

Outline

Stats about inlining ("short-term")
- Why focus on the inliner?
- Inlining of "external" methods
- Inlining of "internal" methods
- Dealing with multiple inlinings of the same callee

Early inlining of anonymous closures ("medium-term")
- AST shapes of interest
- Example 2: `Range.foreach`
- Advantages

Ideas for the future ("long-term")
- Dedicated early-inlining to avoid `NonLocalReturns`
- Inlining using a stackless IR requires zero type-flow analyses

Wrap-up

```
[inliner                231708ms] 68% of compiler run
[inlineException...       7753ms]  2%
[closelim                 4043ms]  1%
[dce                     17837ms]  5%
. . .
[total                  336324ms]
```

Useful distinction:

- External (ie, library) methods that are inlined in methods being compiled
- Methods being compiled that are inlined in methods being compiled

An *external method* is a callee whose ICode is loaded from bytecode.

Inlining of "external" methods:

```
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
   . . .
   15  (0.9%) scala.collection.immutable.Range.foreach
```

Other inlinings (fewer than ten times each): 64

Methods being compiled that were inlined in methods being compiled:

- Times that getters/setters were inlined: 374
- Number of inlined anon-closure `apply()`: **2584** (292 `$sp`).

Breakdown of the rest:

- Each callee inlined at least ten times:

```
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
    . . .
```

- Inlinings for callees inlined fewer than ten times each: 1489

Dealing with multiple inlinings of the same callee.

Example 1: `Range.foreach()`:

- Solution 1: Reformulate to invoke just once (cf. p. 10)

```
@inline final override def foreach[@specialized(Unit) U](f: Int => U) {
  if (length > 0) {
    val last = this.last
    var i = start
    while (i != last) {
      f(i)                          1
      i += step
    }
    f(i)                            2
  }
}
```

- Solution 2, Compiler-supported:
  Share inlined `BasicBlock`s across control paths
  (provided covered by the same exception handlers).
  An extra var can be used to `JUMP` to the right successor
  (`inlineExceptionHandlers` does something similar)

Outline

Stats about inlining ("short-term")
- Why focus on the inliner?
- Inlining of "external" methods
- Inlining of "internal" methods
- Dealing with multiple inlinings of the same callee

Early inlining of anonymous closures ("medium-term")
- AST shapes of interest
- Example 2: `Range.foreach`
- Advantages

Ideas for the future ("long-term")
- Dedicated early-inlining to avoid `NonLocalReturn`s
- Inlining using a stackless IR requires zero type-flow analyses

Wrap-up

Applicability conditions:

1. In some cases, we can know for a callsite what concrete method will be dispatched at runtime.
2. Say, before `uncurry`,
   - for a callsite receiving a `Function` AST node as last argument (anon-closure),
   - where the `Function`'s body is an expression (no `return`) and
   - that argument is used at a single place in the concrete method (to invoke `apply()`. Therefore, the closure doesn't escape).
3. Two cases: we have the AST of the concrete method (“internal”), or bytecode can be loaded (and decompiled into an Scala, not ICode, AST). *BTW, can you live with GOTOs in ASTs?*

Things like: `atOwner`, `withClosed`, etc.
If “all that” holds then . . .

Example 1: `Range.foreach`

```
val rv = <coll>
if (rv.length > 0) {
  val sentinel = rv.last
  var closuVar = rv.start
  var loopCond = true
  while ( loopCond ) {
    <closuBody>
    if(closuVar == sentinel) loopCond = false
    else closuVar += rv.step
  }
}
```

- `rv` is the range instance
- `closuBody` is the original closure body with usages of the closure param substituted with usages of the variable that gets assigned the range’s elements (called “`closuVar`” above).

https://lampsvn.epfl.ch/svn-repos/scala/scala-experimental/trunk/earlyinline

Advantages:

- The “special case” just described takes a heavy load off `Inliner`’s shoulders (and results in a smaller `jar`).
- Early-inlining means less work for other phases, too:
  - `lambdalift`: fewer captured variables, no indirection for them
  - `specialize`
  - faster copy-propagation when eliminating dead closures
- It’s OK to leave untouched those “higher-order callsites” that `inliner` won’t attempt to inline anyway.

Outline

Stats about inlining ("short-term")

- Why focus on the inliner?
- Inlining of "external" methods
- Inlining of "internal" methods
- Dealing with multiple inlinings of the same callee

Early inlining of anonymous closures ("medium-term")

- AST shapes of interest
- Example 2: `Range.foreach`
- Advantages

Ideas for the future ("long-term")

- Dedicated early-inlining to avoid `NonLocalReturn`s
- Inlining using a stackless IR requires zero type-flow analyses

Wrap-up

```
def nonLocalReturnExample(a: Int, b: Int): Boolean = {
 for (i <- 2 to b) if (a % i != 0) return false;
 true
}
```

Currently lowered to:

```
def nonLocalReturnExample(a: Int, b: Int): Boolean = {
  <synthetic> val nonLocalReturnKey1: Object = new Object();
  try {
    scala.this.Predef.intWrapper(2).to(b).foreach[Unit]({
      @SerialVersionUID(0) final <synthetic> class $anonfun
      extends scala.runtime.AbstractFunction1[Int,Unit] with Serializable {
        def this(): anonymous class $anonfun = { $anonfun.super.this(); () };
        final def apply(i: Int): Unit = {
          if (a.%(i).!=(0))
            throw new scala.runtime.NonLocalReturnControl[Boolean(false)](
              nonLocalReturnKey1, false)
          else ()
        }
      };
      (new anonymous class $anonfun(): Int => Unit)
    });
    true
  } catch {
    case (ex @ (_: scala.runtime.NonLocalReturnControl[C._])) =>
      if (ex.key().eq(nonLocalReturnKey1)) ex.value().asInstanceOf[Boolean]()
      else throw ex
  }
}
```

- Without early-inlining but with -optimise
  approx. 170 ICode instructions

```
blocks: [1,7,16,15,26,28,27,25,9,12,10,13,18,19,20,17,11,8,14,22,23,29,24,21,3,4,2,5]

Exception handlers:
  catch (NonLocalReturnControl) in ArrayBuffer(7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17
    consisting of blocks: List(6, 5, 4, 3)
    with finalizer: null
```

- With early-inlining and -optimise
  approx. 65 ICode instructions

```
blocks: [1,2,5,6,9,11,12,13,4]

Exception handlers:
```

Inlining using a stackless IR requires zero type-flow analyses:

- Splicing the CFG of a callee into its caller (both as stackless IR) can be done without worrying about type-stacks at all.
- Conversion into 3-addr and back into expr-language already available (for post-CleanUp Scala ASTs, not ICode):

`http://lamp.epfl.ch/~magarcia/ScalaCompilerCornerReloaded/2011Q4/PartialEval3A.pdf`

- In that prototype, an "`if`" looks visually nested, e.g.

```
if(c1) {
  if(c2) {
    stmt;
  }
}
```

- but there's a one step desugaring to "truly" CFG-based stackless IRs (your choice of SSA or three-address code)

```
If(Ident(c1)) GOTO(label)
```

Outline

Stats about inlining ("short-term")
- Why focus on the inliner?
- Inlining of "external" methods
- Inlining of "internal" methods
- Dealing with multiple inlinings of the same callee

Early inlining of anonymous closures ("medium-term")
- AST shapes of interest
- Example 2: `Range.foreach`
- Advantages

Ideas for the future ("long-term")
- Dedicated early-inlining to avoid `NonLocalReturn`s
- Inlining using a stackless IR requires zero type-flow analyses

Wrap-up

It's hard to pick *just one* of the options below because both stand to benefit *all* Scala programs . . .

1. Improving the current `Inliner`. Details at

   `http://lamp.epfl.ch/~magarcia/ScalaCompilerCornerReloaded/2011Q4/Inliner.pdf`

2. Early inlining of anonymous closures

The next one requires brainstorming, planning, and some knowledge of McGill's Soot (i.e., most likely a master thesis):

3. Optimizations based on stackless IR