What the optimizer does to your code

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

2012-04-18

Outline

Things the optimizer is good at

- Example
- Pros and Cons

Ongoing and Future work

- Early inlining
- Parallelizing an optimization phase

Further information

The optimizer strives to “proceduralize” code patterns of the form:

1. instantiation of anonymous-closure class — “`A`”
2. monadic call (`foreach`, `filter`, etc) with `A` argument — “`M`”
   Note: for JIT purposes, this callsite may or may not be hot
3. application of `A` in the callee’s body *(usually, inside a loop)*

The above results from rephrasing AST function nodes via OO

- Example:

```
var captured = 123
for(i <- 1 to 10) { Console.print(xs(i) + captured) }
```

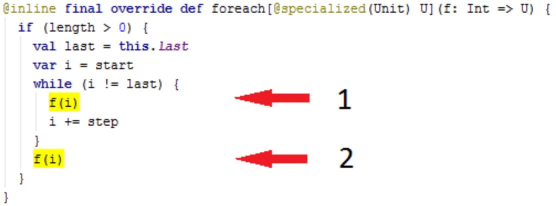
- What the optimizer gets to see (*simplified*)

```
var captured: Int = 123;
/*- `foreach` invocation on Range */
scala.Predef.intWrapper(1).to(10).foreach[Unit]({

  /*- class definition local to block expression */
  final class $anonfun
  extends scala.runtime.AbstractFunction1[Int,Unit]
  with Serializable {
    /*- argless constructor omitted */
    def apply(i: Int) { Console.print(xs(i) + captured) }
  } // end of class $anonfun

  (new $anonfun()) /*- argument to `foreach` */
})
```

- Resulting while loop (*excerpt*)

```
79: iload 9          /* loop condition */
81: iload 6
83: if_icmpne   87  /* iterate */
86: return
87: getstatic   #50; //Field scala/Console$.MODULE$:Lscala/Console$;
90: new   #52; //class scala/collection/mutable/StringBuilder
. . .
135: goto 79 /* backedge starts here */
```

- Pros: fewer classes (inlined closures can be removed)
- Cons: as with all inlining, code duplication



```
@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)
      i += step
    }
    f(i)
  }
}
```

Tracing the inliner's reasoning: `-Ylog:inliner -Ydebug`

Closure elimination alone is not enough. Example:

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

```
def nonLocalReturnExample(a: Int, b: Int): Boolean = {
  val retKey = new Object();
  try {
    scala.Predef.intWrapper(2).to(b).foreach[Unit]({
      final class $anonfun extends AbstractFunction1[Int,Unit] {
        def apply(i: Int) {
          if (a.%(i).!=(0))
            throw new NonLocalReturnControl(retKey, false)
            /*- `return false` would quit `apply()` only */
        }
      }; (new $anonfun()) });
    true
  } catch { case (ex @ (_: NonLocalReturnControl)) =>
          if (ex.key eq retKey) ex.value.asInstanceOf[Boolean]
          else throw ex
  }
}
```

And now the fine print:

Given a callsite receiving a `Function` AST node as last argument (anon-closure), early inlining is feasible when:

- the callee to dispatch at runtime is known statically,
- the argument is used at most once in the concrete method (to invoke `apply()`, ie. no excessive code duplication).

Two cases:

- the AST of the concrete method is being compiled, or
- bytecode can be loaded (and decompiled into an Scala AST).

Simpler CFG, instruction count halved, no exception handling

Dead-code elimination focuses on a single method at a time.

A recipe for task parallelism:

- Work items are queued in a `java.util.concurrent.PriorityBlockingQueue`
- Larger methods processed first (for load balancing)
- “No more work” is signalled by poison pills

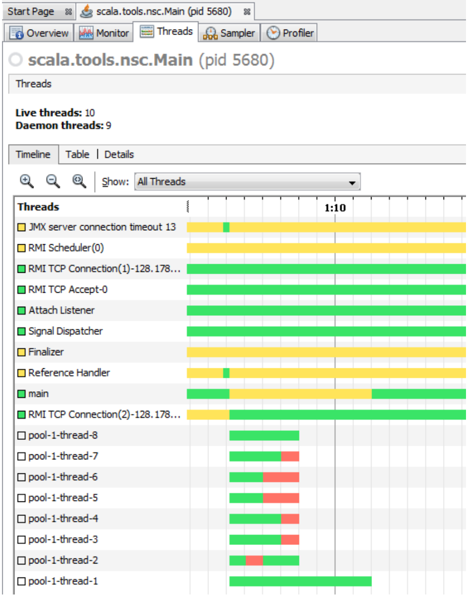
```
// once the queue is full ...
val exec = java.util.concurrent.Executors.newFixedThreadPool(MAX_THRE
val workers =
  for(i <- 1 to MAX_THREADS)
  yield { val t = new DCETask(q, poison); exec.execute(t); t }
workers foreach { w => q put poison }
exec.shutdown()
while(!exec.isTerminated) {
  exec.awaitTermination(1, TimeUnit.MILLISECONDS)
}
assert(q.isEmpty)
```

Well, let’s not forget about synchronization:

- make mutable-shared-state not shared across threads (e.g., `Linearizer` and `Peephole` are now instance-level and thus not shared across tasks submitted to `Executor`)
- now the tricky part. Lock all accesses to the typer, i.e. calls `Tree.tpe` or `Symbol.info`

```
private def getProduced(i: Instruction): Int = {
  if(i.isInstanceOf[opcodes.CALL_METHOD]) {
    /*- CALL_METHOD.produced() calls producedType */
    global synchronized i.produced
  } else i.produced
}
```

With 8 threads,
3x speedup
(additional threads
are useless,
due to contention
on `typer`).



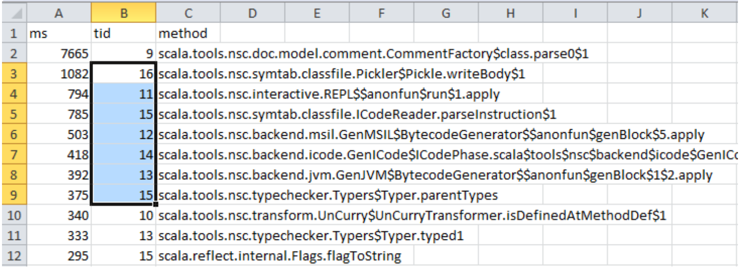
scala.tools.nsc.Main (pid 5680)

Threads

Live threads: 10
Daemon threads: 9

Threads
□ JMX server connection timeout 13
□ RMI Scheduler(0)
□ RMI TCP Connection(1)-128.178...
□ RMI TCP Accept-0
□ Attach Listener
□ Signal Dispatcher
□ Finalizer
□ Reference Handler
□ main
□ RMI TCP Connection(2)-128.178...
□ pool-1-thread-8
□ pool-1-thread-7
□ pool-1-thread-6
□ pool-1-thread-5
□ pool-1-thread-4
□ pool-1-thread-3
□ pool-1-thread-2
□ pool-1-thread-1

Load-balancing and all, there can be and there are outliers:



	A	B	C	D	E	F	G	H	I	J	K
1	ms	tid	method								
2	7665	9	scala.tools.nsc.doc.model.comment.CommentFactory$class.parse0$1								
3	1082	16	scala.tools.nsc.symtab.classfile.Pickler$Pickle.writeBody$1								
4	794	11	scala.tools.nsc.interactive.REPL$$anonfunrun1.apply								
5	785	15	scala.tools.nsc.symtab.classfile.ICodeReader.parseInstruction$1								
6	503	12	scala.tools.nsc.backend.msil.GenMSIL$BytecodeGenerator$$anonfun$genBlock$5.apply								
7	418	14	scala.tools.nsc.backend.icode.GenICode$ICodePhase.scala$tools$nsc$backend$icode$GenICo								
8	392	13	scala.tools.nsc.backend.jvm.GenJVM$BytecodeGenerator$$anonfun$genBlock$1$2.apply								
9	375	15	scala.tools.nsc.typechecker.Typers$Typer.parentTypes								
10	340	10	scala.tools.nsc.transform.UnCurry$UnCurryTransformer.isDefinedAtMethodDef$1								
11	333	13	scala.tools.nsc.typechecker.Typers$Typer.typed1								
12	295	15	scala.reflect.internal.Flags.flagToString								

A single work-unit (top line) holds its poor worker busy, even after all other workers are done and sit idle.

Summing up:

- 2.10 includes a significantly faster optimizer
- Improvements on the way (early inlining, parallel optimizer)
- Longer term, candidate ideas for more radical improvements (three-address code, effects analysis, runtime monomorphization)

http://lampwww.epfl.ch/~magarcia/ScalaCompilerCornerReloaded/