`ClosureElimination` and `DeadCodeElimination`

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

December 4th, 2011

Abstract

`ClosureElimination` [1, §3.3.2] and `DeadCodeElimination` [1, §3.3.3] work synergistically after the `inliner` phase to:

1. replace an instruction (that accesses data from a location which in turn holds a value copied from another) by a direct access to the original value (Sec. 1)
2. recursively mark certain instructions as useful to later discard unmarked ones (Sec. 2).

Both phases above rely on a peephole optimizer (Sec. 3) to simplify consecutive ICode instructions, where both the original and the resulting instructions leave the operand stack unchanged.

Associated to each optimization pass there's a dataflow analysis (DFA):

- for `inliner` [2] and `inlineExceptionHandlers` [3] it's `MethodTFA`,
- for `ClosureElimination` it's `ReachingDefinitionsAnalysis`,
- for `DeadCodeElimination` it's `CopyAnalysis`, and
- for the peephole pass it's `LivenessAnalysis`.

An introduction to the DFA infrastructure can be found in [4].

```
           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
                      . . .
              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 Last link in a chain of data accesses replaced by direct access ("ClosureElimination")

A "chain of data accesses" refers to a assignments of the form (not necesarily consecutive):

```
v1  = v
. . .
v2  = v1
. . .
v99 = v98
. . .
<usage of v99>
```

These "multi-hop" accesses result from inlining `apply()` callees. Frequently, those `apply()` correspond to local closures, which necessarily access part of their environment through an outer instance, other fields in the closure-class, and "conveyor params" passed to its constructor or to the `apply()` invocation (this magic is provided by `lambdalift`, [5]). A local closure-class has at least one field (for the outer instance) that may lead after inlining to the access-chains just described (after inlining, that same value is available as "`this`").

Operationally, this phase visits each ICode instruction in a method, focusing on three cases:

```
case LOAD_LOCAL(l) if info.bindings isDefinedAt LocalVar(l) =>
  , , ,

case LOAD_FIELD(f, false) /* if accessible(f, m.symbol) */ =>
  , , ,

case UNBOX(_) =>
  , , ,
```

For example, a `LOAD_LOCAL` may be replaced with another. The decision to replace a `LOAD_LOCAL` or `LOAD_FIELD` instruction is made based on copy-propagation results (computed by a `CopyAnalysis`, Sec. 1.2). Details about doing away with `UNBOX` in Sec. 1.1.

Let's pause for a moment to consider what's in a name: `ClosureElimination`.

1. This phase performs no rewriting beyond what's described above, and thus by itself doesn't elide any closures. That will have to wait for:
   (a) `DeadCodeElimination` (Sec. 2) to record those closure-classes that are instantiated at least once.
   (b) Afterwards, `GenJVM` (resp. `GenMSIL`) will elide unused closure-classes:

   ```
   if (settings.Xdce.value)
     for ( (sym, cls) <- icodes.classes
           if inliner.isClosureClass(sym) && !deadCode.liveClosures(sym))
       icodes.classes -= sym
   ```

2. A direct access makes redundant those locations after the first link in an access-chain. These redundant locations will be eliminated along with instructions deemed useless during `DeadCodeElimination` (Sec. 2).

1.1 Eliding unboxing

Upon visiting an `UNBOX` instruction, copy-propagation may inform that some local-var holds at that program-point the unboxed counterpart for the boxed value on top of the stack. This situation is signalled by an abstract value of the form `Boxed(LocalVar(extraVarAddedByGenICode))` on which the `UNBOX` operates, where `extraVarAddedByGenICode` contains the unboxed value. In this case, the `UNBOX` is replaced with the instruction sequence `DROP; LOAD_LOCAL(extraVarAddedByGenICode)`.

The `extraVarAddedByGenICode` variable was introduced just for that purpose back in `GenICode`. Quoting from `genLoad()`:

```
case Apply(fun @ _, List(expr)) if (definitions.isBox(fun.symbol)) =>
  debuglog("BOX : " + fun.symbol.fullName);
  val ctx1 = genLoad(expr, ctx, toTypeKind(expr.tpe))
  val nativeKind = toTypeKind(expr.tpe)
  if (settings.Xdce.value) {
    /*- we store this boxed value to a local, even if not really needed.
        boxing optimization might use it, and dead code elimination will
        take care of unnecessary stores */
    var loc1 = ctx.makeLocal(tree.pos, expr.tpe, "boxed")
    ctx1.bb.emit(STORE_LOCAL(loc1))
    ctx1.bb.emit(LOAD_LOCAL(loc1))
  }
  ctx1.bb.emit(BOX(nativeKind), expr.pos)
  generatedType = toTypeKind(fun.symbol.tpe.resultType)
  ctx1
```

By now we have (possibly in different `BasicBlocks`):

```
<original instructions to load unboxed value>
STORE_LOCAL extraVarAddedByGenICode
LOAD_LOCAL extraVarAddedByGenICode
BOX
. . .
// there used to be an UNBOX here that was elided in ClosureElimination
DROP
LOAD_LOCAL extraVarAddedByGenICode
```

The subsequent `DeadCodeElimination` pass will elide the redundant instructions. The peephole pass changes nothing in this case.

```
TODO

Which are those redundant instructions? (a) or (b)?
  (a) BOX; DROP; LOAD_LOCAL
  (b) LOAD_LOCAL; BOX; DROP;
```

1.2 `CopyAnalysis`

Described in [4].

2 Recursive marking of useful instructions, eliding of the rest (“`DeadCodeElimination`”)

This phase realizes a “mark & sweep” algorithm consisting of:

- iterative marking of "useful" instructions in a method, which in turn triggers marking as "useful" those instructions which are reaching-definitions to a previously marked instruction.
- Along the way, those closure-classes instantiated at least once are tracked as "`liveClosures`".

```
case nw @ NEW(REFERENCE(sym)) =>

  assert(nw.init ne null,
         "null new.init at: " + bb + ": " + idx + "(" + instr + ")")

  worklist += findInstruction(bb, nw.init)

  if (inliner.isClosureClass(sym)) {
    liveClosures += sym
  }
```

- Afterwards, `sweep(IMethod)` leaves out all non-marked instructions.

The `dieCodeDie(IMethod)` comprises three sub-steps (Sec. 2.2 to Sec. 2.4) each massaging some maps for the next.

2.1 Auxiliary maps

- `var defs: immutable.Map[(BasicBlock, Int), immutable.Set[rdef.lattice.Definition]]`
  populated during `collectRDef()`, and queried during `mark()`. Its keys denote `LOAD_LOCAL(var)` instructions, the corresponding image lists for each local (including but not only `var`) an over-approximation of the set of instructions defining the local's value.
- `val worklist: mutable.Set[(BasicBlock, Int)]`
  initially populated during `collectRDef()`, afterwards `mark()` removes and adds instructions until it becomes empty (not really a fix-point because the sequencing of `collectRDef()`, `mark()`, and `sweep()` does not constitute an iterative dataflow, with lattice and such. It's monotone all right, but it does not extend `DataFlowAnalysis`).
- `val useful: mutable.Map[BasicBlock, mutable.BitSet]`
  end-station for instructions that will survive `sweep()`
- `var accessedLocals: List[Local]`
  collects those variables having a useful `LOAD_LOCAL` or `STORE_LOCAL`, as well as all params. It's used to replace in `IMethod` the incoming var declarations with those actually used in the method (this is done just before leaving `dieCodeDie(IMethod)`).

2.2 Seeding the list of useful instructions ("`collectRDef(m)`")

This first sub-step seeds a `worklist` with instructions (any instruction added here will be classified as `useful` right away by `mark()`, so apply judgement). The following instructions are always added, irrespective of their control-flow relevance:

```
case RETURN(_) | JUMP(_) | CJUMP(_, _, _, _) | CZJUMP(_, _, _, _) | STORE_FIELD(_, _) |
    THROW(_)  | LOAD_ARRAY_ITEM(_) | STORE_ARRAY_ITEM(_) | SCOPE_ENTER(_) | SCOPE_EXIT(_) | STORE_THIS(_) |
    LOAD_EXCEPTION(_) | SWITCH(_, _) | MONITOR_ENTER() | MONITOR_EXIT()
  => worklist += ((bb, idx))

case CALL_METHOD(m1, _) if isSideEffecting(m1) =>
  worklist += ((bb, idx)); log("marking " + m1)

case CALL_METHOD(m1, SuperCall(_)) =>
  worklist += ((bb, idx)) // super calls to constructor
```

```
TODO

  That's weird (marking as useful irrespective of control-flow relevance).
  What if they are dead code?
```

- At this point, a `DROP` is considered useful if (a) any of its "reaching-defs" is a potentially side-effecting method invocation; or (b) the `DROP` is part of the prolog of an exception handler; or (c) it drops a dup-ed value or a reference to a module:

```
  case CALL_METHOD(m1, _) if isSideEffecting(m1) => true
  case LOAD_EXCEPTION(_) | DUP(_) | LOAD_MODULE(_) => true
```

```
TODO

   Question: In the snippet below,
             is "rd.stack.head" a synonym for "rdef.findDefs(bb, idx, 1)" ?
             (the former is easier to understand)

    case DROP(_) =>
      val necessary = rdef.findDefs(bb, idx, 1) exists { p =>
```

2.3 `mark()`

This sub-step is all about moving one instruction at a time from `worklist` to `useful`, adding its dependencies to `worklist`, until it's empty. Some notes:

- Tagging as useful an instruction leads to making useful any forthcoming `DROP` of the value it pushes:

```
  useful(bb) += idx
  dropOf.get(bb, idx) foreach {
      for ((bb1, idx1) <- _)
        useful(bb1) += idx1
  }
```

- This is where an anon-closure escapes elision:

```
  case nw @ NEW(REFERENCE(sym)) =>
  assert(nw.init ne null, "null new.init at: " + bb + ": " + idx + "(" + instr + ")")
  worklist += findInstruction(bb, nw.init)
```

```
  if (inliner.isClosureClass(sym)) {
    liveClosures += sym
  }
```

```
TODO Adapt for Scala.Net (NEW.init vs. <init> on CLR)
```

- The general case just adds all (not yet useful) instructions that potentially pushed values the instruction consumes:

```
case _ =>
  for ((bb1, idx1) <- rdef.findDefs(bb, idx, instr.consumed) if !useful(bb1)(idx1)) {
    worklist += ((bb1, idx1))
  }
```

2.4 `sweep(m)`

From the auxiliary data structures, only `useful` and `accessedLocals` are used in this last sub-step. Before eliding proper, `computeCompensations(IMethod)` iterates over all instructions waiting to be discarded ("`!useful(bb)(idx)`"). These instructions obviously won't consume anything from the stack, thus the instructions loading such values are adorned with a `DROP(consumedType)` (these reaching-defs might well be non-useful themselves, in which case the just added compensation won't be emitted).

Sidenote: For a `NEW` reaching-def, the `DROP` is attached to the init-invocation, because the ICode being optimized 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
```

With `compensations` computed, it's time to clear the `BasicBlock`'s instructions and emit only the useful ones (along with any compensations). Additionally, only used variables will be kept:

```
i match {
  case LOAD_LOCAL(l) if !l.arg =>
    accessedLocals = l :: accessedLocals
  case STORE_LOCAL(l) if !l.arg =>
    accessedLocals = l :: accessedLocals
  case _ => ()
}
```

```
TODO A more informative return type for computeCompensations() is:

  collection.Map[(BasicBlock, Int), List[DROP]]
```

Listing 1: Sec. 3

```
case (STORE_LOCAL(x), LOAD_LOCAL(y)) if (x == y) =>
  var liveOut = liveness.out(bb)
  if (!liveOut(x)) {
    log("store/load to a dead local? " + x)
    val instrs = bb.getArray
    var idx = instrs.length - 1
    while (idx > 0 && (instrs(idx) ne i2)) {
      liveOut = liveness.interpret(liveOut, instrs(idx))
      idx -= 1
    }
    if (!liveOut(x)) {
      log("removing dead store/load " + x)
      Some(Nil) /*- Outcome 1 of 3: Dead var, Eliding */
    } else None /*- Outcome 2 of 3: Alive var, leave as-is */
  } else
    Some(List(DUP(x.kind), STORE_LOCAL(x))) /*- Outcome 3 of 3: Cosmetic */
```

2.5 `ReachingDefinitionsAnalysis`

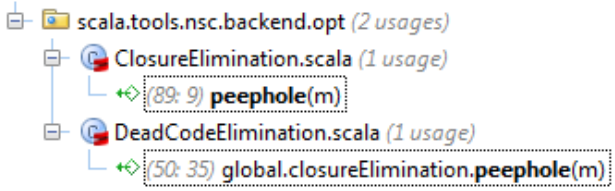
Described in [4].

3 Peephole optimizations

This pass considers just two consecutive instructions at a time and may peform a simple replacement, in that case both the original and the resulting instructions leave the operand stack unchanged.

```
def peep(bb: BasicBlock, i1: Instruction, i2: Instruction) = (i1, i2) match {
  . . .
```

It is invoked once other optimization passes are done with a method:



Just one of the `peephole` rewritings relies on a liveness analysis (Sec. 3.1), as the snippet in Listing 1 shows. In short, one of two rewritings is performed, the second cosmetic. Deciding between both involves traversing the block's instructions starting with the last one, because `LivenessAnalysis` is a backward data-flow analysis.

```
TODO

   Outcome 3 seems better than Outcome 2 (hey, it is even cosmetic!)
   Why is Outcome 2 used instead?
```

3.1 `LivenessAnalysis`

Described in [4].

A Appendix: Comparison with McGill's Soot

There's a stackless (three-address) IR for the Scala compiler ("`imp`", [6]) that is also converted back to stack-language as described in [7]. Soot's Jimple supports several optimizations (listed below) that as of now `imp` lacks. Quoting from `http://www.sable.mcgill.ca/soot/tutorial/phase/phase.html`

> *Jimple Body Creation (jb):*
> *creates a `JimpleBody` for each input method, using either `coffi`, to read `.class` files, or the jimple parser, to read `.jimple` files.*
>
> 1. *The Local Splitter identifies DU-UD webs for local variables and introduces new variables so that each disjoint web is associated with a single local.*
> 2. *The Jimple Local Aggregator removes some unnecessary copies by combining local variables. Essentially, it finds definitions which have only a single use and, if it is safe to do so, removes the original definition after replacing the use with the definition's right-hand side. At this stage in JimpleBody construction, local aggregation serves largely to remove the copies to and from stack variables which simulate load and store instructions in the original bytecode.*
> 3. *Unused Local Eliminator*
> 4. *The Type Assigner gives local variables types which will accommodate the values stored in them over the course of the method.*
> 5. *Copy Propagator: This phase performs cascaded copy propagation. If the propagator encounters situations of the form:*
>
> ```
> A: a = ...;
>   ...
> B:  x = a;
>   ...
> C: ... = ... x;
> ```
>
> *where `a` and `x` are each defined only once (at `A` and `B`, respectively), then it can propagate immediately without checking between `B` and `C` for redefinitions of `a`. In this case the propagator is global. Otherwise, if a has multiple definitions then the propagator checks for redefinitions and propagates copies only within extended basic blocks.*
>
> 6. *The Dead Assignment Eliminator eliminates assignment statements to locals whose values are not subsequently used, unless evaluating the right-hand side of the assignment may cause side-effects.*
> 7. *Post-copy propagation Unused Local Eliminator (`jb.cp-ule`). Removes any locals that are unused after copy propagation.*

8. *The Local Packer attempts to minimize the number of local variables in a method by reusing the same variable for disjoint DU-UD webs. Conceptually, it is the inverse of the Local Splitter.*
9. *The Nop Eliminator removes nop statements from the method.*
10. *The Unreachable Code Eliminator removes unreachable code and traps whose catch blocks are empty.*
11. *The Trap Tightener changes the area protected by each exception handler, so that it begins with the first instruction in the old protected area which is actually capable of throwing an exception caught by the handler, and ends just after the last instruction in the old protected area which can throw an exception caught by the handler. This reduces the chance of producing unverifiable code after pruning exceptional control flow within CFGs.*

TODO Compare McGill*'s Soot and ProGuard for dead-code elimination.*

References

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

[2] Miguel Garcia. ICode inlining, 2011. Notes at *The Scala Compiler Corner*. http://lamp.epfl.ch/~magarcia/ScalaCompilerCornerReloaded/2011Q4/Inliner.pdf.

[3] Miguel Garcia. InlineExceptionHandlersPhase, 2011. Notes at *The Scala Compiler Corner*. http://lamp.epfl.ch/~magarcia/ScalaCompilerCornerReloaded/2011Q4/InlineExceptHandler.pdf.

[4] Miguel Garcia. Introduction to Dataflow Analyses on ICode, 2011. Notes at *The Scala Compiler Corner*. http://lamp.epfl.ch/~magarcia/ScalaCompilerCornerReloaded/2011Q4/DFAICode.pdf.

[5] Miguel Garcia. Lambda Lifting, 2011. Notes at *The Scala Compiler Corner*. http://lamp.epfl.ch/~magarcia/ScalaCompilerCornerReloaded/2011Q2/LambdaLift.pdf.

[6] Miguel Garcia. Moving Scala ASTs one step closer to C, 2011. Notes at *The Scala Compiler Corner*. http://lamp.epfl.ch/~magarcia/ScalaCompilerCornerReloaded/2011Q2/Moving3A.pdf.

[7] Miguel Garcia. Partial evaluation based on three-address-form, including conversion back to stack-based language, 2011. Notes at *The Scala Compiler Corner*. http://lamp.epfl.ch/~magarcia/ScalaCompilerCornerReloaded/2011Q4/PartialEval3A.pdf.