Dataflow Analyses on ICode

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

November 28th, 2011

Abstract

These notes provide an introduction to the DFA (data-flow analysis) infrastructure for ICode, as used by the ICode optimization phases [1] in the Scala compiler.

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

- for `inliner` [2] and `inlineExceptionHandlers` [3] it's `MethodTFA`,
- for `ClosureElimination` [4] it's `ReachingDefinitions`,
- for `DeadCodeElimination` [4] it's `CopyAnalysis`, and
- for the peephole pass [4] it's `LivenessAnalysis`.

The write-ups referenced above cover the role of these analyses in the context of each optimization.

Contents

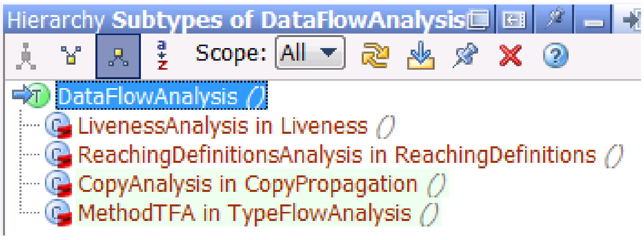


Figure 1: Dataflow analyses for ICode

Listing 1: `typeStackLattice`

```
/** The lattice of type stacks. It is a straightforward extension of
 *  the type lattice (lub is pairwise lub of the list elements).
 */
object typeStackLattice extends CompleteLattice {
  import icodes._
  type Elem = TypeStack

  override val top    = new TypeStack
  override val bottom = new TypeStack
  val exceptionHandlerStack: TypeStack = new TypeStack(List(REFERENCE(definitions.AnyRefClass)))

  def lub2(exceptional: Boolean)(s1: TypeStack, s2: TypeStack) = {
    if (s1 eq bottom) s2
    else if (s2 eq bottom) s1
    else if ((s1 eq exceptionHandlerStack) || (s2 eq exceptionHandlerStack))
      Predef.error("merging with exhan stack")
    else {
      new TypeStack((s1.types, s2.types).zipped map icodes.lub)
    }
  }
}
```

1 Overview

Four analyses are provided out-of-the-box for ICode (Figure 1) and more can be defined by choosing a lattice and a transfer function. A fix-point will be searched by an iterative approach [5, §8.4] (backward or forward, depending on whether "`backwardAnalysis(blockTransfer)`" or "`forwardAnalysis(blockTransfer)`" is invoked in the `run()` override).

1.1 An example of typeflow analysis on basic blocks

Out-of-the-box, a `TypeStack` lattice is available (Listing 1). The level of granularity considered by the `MethodTFA` analysis is not `Instruction` but `BasicBlock`. In other words:

- `trait DataFlowAnalysis[L <: CompleteLattice]`
  requires a `type P <: ProgramPoint[P]`, and
- `MethodTFA` considers `type P = BasicBlock`

The contract of `ProgramPoint` is:

Listing 2: `jimpleTest` in Scala

```
object jimpleTest {

class A {}
class B extends A {}
class C extends A {}

  def main(args : Array[String]) {
    val x = if (java.lang.System.currentTimeMillis() == 0)
             new B
           else
             new C
  }
}
```

```
trait ProgramPoint[a <: ProgramPoint[a]] {
  def predecessors: List[a]
  def successors: List[a]
  def exceptionHandlerStart: Boolean
}
```

Typeflow one ICode instruction at a time:

```
    /** Abstract interpretation for one instruction. */
    def interpret(in: typeFlowLattice.Elem, i: Instruction): typeFlowLattice.Elem
```

The example in Listing 2 computes two values (of different types, in two different control-flow paths) for the same variable. The ICode output (obtained via `-Xprint:icode`) is depicted in Listing 3.

Both of blocks 2 and 3 have block 4 as successor, which expects a typestack consisting of `REFERENCE(jimpleTest.A)` on entry, while the outgoing typestacks of 2 and 3 are of the form `REFERENCE(jimpleTest.B)` and `REFERENCE(jimpleTest.C)` resp.

Without writing a compiler plugin, one can see `MethodTFA` in action by debugging the compiler in a run with `-Yinline`. That activates `analyzeMethod` in `Inliners` which shows the steps to run the typestackflow analysis:

```
val tfa = new analysis.MethodTFA();
tfa.init(m)
tfa.run
for (bb <- linearizer.linearize(m)) {

  // check tfa.in(bb) and tfa.out(bb)

  // for (i <- bb) iterates the instructions in the basic block bb,
  // info = tfa.interpret(info, i) can be invoked
}
```

For the `typeFlowLattice`, its elements are of the form `IState[VarBinding, icodes.TypeStack]`.

- The first type param stands for an environment (where each binding associates a local var with its `TypeKind`), except that a `VarBinding` map returns bottom (i.e., `typeLattice.bottom` not `typeFlowLattice.bottom`) for vars not

Listing 3: `jimpleTest` in ICode, Sec. 1.1

```
object jimpleTest extends java.lang.Object, ScalaObject {
  // fields:

  // methods
  def <init>(): object jimpleTest { . . . }
  Exception handlers:

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

  1:
    8  CALL_METHOD java.lang.Systemjava.lang.System.currentTimeMillis (static-class)
    8  CONSTANT (Constant(0))
    8  CJUMP (LONG)EQ ? 2 : 3

  2:
    9  NEW REFERENCE(jimpleTest$B)
    9  DUP
    9  CALL_METHOD jimpleTest$BjimpleTest$B.<init> (static-instance)
    9  JUMP 4

  3:
    11  NEW REFERENCE(jimpleTest$C)
    11  DUP
    11  CALL_METHOD jimpleTest$CjimpleTest$C.<init> (static-instance)
    8  JUMP 4

  4:
    8  STORE_LOCAL value x
    8  SCOPE_ENTER value x
    8  SCOPE_EXIT value x
    8  RETURN (UNIT)

  }
  Exception handlers:
}
```

yet in the map.

- The second type param (`TypeStack`) is just a stack of `TypeKind` elements.

```
/** A map which returns the bottom type for unfound elements */
class VarBinding extends mutable.HashMap[icodes.Local, icodes.TypeKind] {
  override def get(l: icodes.Local) = super.get(l) match {
    case Some(t) => Some(t)
    case None   => Some(typeLattice.bottom)
  }

  def this(o: VarBinding) = {
    this()
    this ++= o
  }
}
```

Listing 4: Sec. 1.2

```
method: InterfaceDemo.hardest
 block: 1
        type stack : []
        no reaching-defs on the empty stack
  0|  CALL_METHOD java.lang.Systemjava.lang.System.currentTimeMillis (static-class)
        type stack : [LONG]
          reaching the slot at depth: 0
            def: /CALL_METHOD java.lang.Systemjava.lang.System.currentTimeMillis (static-class) in block 1\
  1|  CONSTANT (Constant(0))
        type stack : [LONG,LONG]
          reaching the slot at depth: 0
            def: /CONSTANT (Constant(0)) in block 1\
          reaching the slot at depth: 1
            def: /CALL_METHOD java.lang.Systemjava.lang.System.currentTimeMillis (static-class) in block 1\
  2|  CJUMP (LONG)EQ ? 2 : 3
        last type stack : []
          no reaching-defs on the empty stack
```

1.2 An example of reaching definitions

`ReachingDefinitionsAnalysis` gives for each stack slot the instruction(s) that have written the slot. For example, the following instructions:

```
method: InterfaceDemo.hardest
 block: 1
        type stack : []
  0|  CALL_METHOD java.lang.Systemjava.lang.System.currentTimeMillis (static-class)
        type stack : [LONG]
  1|  CONSTANT (Constant(0))
        type stack : [LONG,LONG]
  2|  CJUMP (LONG)EQ ? 2 : 3
        last type stack : []
```

make the traversal of reaching-defs report the first instruction twice (until its stack slot gets popped). That instruction appears in Listing 4 (as definition) first with depth 0 (i.e., the value is on top) and after the push by `CONSTANT` with depth 1. The traversal was performed in the "natural" way:

```
/** Prints as a table the defs reaching instrIdx. */
def makeSenseOfReachingDefs(rdefVars: Set[(Local, BasicBlock, Int)],
                            rdefStack: List[Set[(BasicBlock, Int)]],
                            instrIdx: Int) {
  if(rdefStack.isEmpty){
     scala.Console.println("\t\t\t no reaching-defs on the empty stack")
  }
  for((slot, depth) <- rdefStack.zipWithIndex){
    scala.Console.println("\t\t\t reaching the slot at depth: " + depth)
    for((bb, bbidx) <- slot) {
      scala.Console.println("\t\t\t\t def: " + where(bb, bbidx))
    }
  }
  def where(b: BasicBlock, i: Int) = "/" + b(i) + " in block " + b + "\\"
}
```

2 ReachingDefinitionsAnalysis

2.1 Lattice

As with all DFAs, it's best to look first at the lattice that the reaching-def analysis adopts, as given by `rdefLattice.Elem`

```
type Elem = IState[
                    Set [ (Local, BasicBlock, Int) ],
                    List[ Set[ (BasicBlock, Int) ] ]
                  ]
```

- The abstract state of variables is represented as a set of triples, where different triples may include the same `Local`. To illustrate, the same information could be represented as:

  - `Map [ Local, Set[ (BasicBlock, Int) ] ]`, or
  - `MultiMap [ Local, (BasicBlock, Int) ]`

- The abstract state of the operand stack includes for each stack position a set of ICode program locations. List head is stack top.

`rdefLattice.lub2()` computes the entry abstract stack for an exception handler somewhat differently as compared to its `MethodTFA` counterpart. Here's how `typeFlowLattice.lub2()` handles that:

```
  val stack =
    if (exceptional) typeStackLattice.exceptionHandlerStack
    else typeStackLattice.lub2(exceptional)(a.stack, b.stack)
```

In contrast, `rdefLattice.lub2()` does not special-case exception handlers:

```
if (a.stack.isEmpty) b.stack
else if (b.stack.isEmpty) a.stack
else {
  (a.stack, b.stack).zipped map (_ ++ _)
}
```

Instead, that's handled via interplay with the instruction-transfer-function:

```
def interpret(b: BasicBlock, idx: Int, in: lattice.Elem): Elem
```

where it can *clearly* be read:

```
  instr match {
    case STORE_LOCAL(l1) =>
      locals = updateReachingDefinition(b, idx, locals)
      stack = stack.drop(instr.consumed)
    case LOAD_EXCEPTION(_) => /*- here's where the abstract-stack for exception handlers */
      stack = Nil
    case _ =>
      stack = stack.drop(instr.consumed)
  }
```

```
TODO
  Why ``stack = Nil'' instead of loading, say,
  the set of all instruction-positions of all blocks covered by the handler?
  (this information can be grabbed via BasicBlock.method.exh)

  Alternatively, rather than all those instructions,
    a distinguished representative can be used.

  Alternatively, mutable.Map[BasicBlock, mutable.BitSet] may be compact enough.

Also related, in init()

  m.exh foreach { e =>
    in(e.startBlock) = lattice.IState(new ListSet[Definition], List(new StackPos))
  }
```

2.2 Initialization

Two helper functions iterate over the instructions on each `BasicBlock` to populate the following block-level summaries:

- `gen`: last assignments per block
  `Map[BasicBlock, Set[ (Local, BasicBlock, Int) ]]`
- `kill`: variables assigned at least once, per block
  `Map[BasicBlock, Set[Local]]`
- `drops`: how many more elements are popped than pushed
  `Map[BasicBlock, Int]`
- `outStack`: net growth in the stack contributed by this block
  `Map[BasicBlock, List[ Set[(BasicBlock, Int)] ] ]`

2.3 Block-level and instruction-level transfer functions

In order to compute the abstract state (for local variables and for the operand stack), the block-level transfer function just looks up info prepared by `init()` (Sec. 2.2).

- The abstract state for local variables
  - trims previous definitions for those variables assigned in the current basic block (i.e., "`kill(b)`"),
  - keeps reaching-defs for variables not assigned, and
  - includes a more recent reaching-definition for each variable assigned at least once.
- The abstract stack (on basic block exit) adds on top of the incoming stack (with its top `drops(b)` elements chopped off, as they are consumed in the basic block) the net elements pushed by `b` (that delta is given by `outStack(b)`):

```
private def blockTransfer(b: BasicBlock, in: lattice.Elem): lattice.Elem = {
  var locals: ListSet[Definition] = (in.vars filter { case (l, _, _) => !kill(b)(l) }) ++ gen(b)
  if (locals eq lattice.bottom.vars) locals = new ListSet[Definition]
  IState(locals, outStack(b) ::: in.stack.drop(drops(b)))
}
```

In other words, unlike its `MethodTFA` counterpart, `ReachingDefinitionsAnalysis.blockTransfer()` does not use the instruction-level transfer function.

Regarding the instruction-level transfer function, it intercepts `STORE_LOCAL` instructions to update the state of variables, and pops and pushes (instruction-positions) as given by `instr.consumed` and `instr.produced` (except for `LOAD_EXCEPTION`, Sec. 2.1).

```
TODO check via assert, in rdefLattice.lub2():

  // !!! These stacks are with some frequency not of the same size.
  // I can't reverse engineer the logic well enough to say whether this
  // indicates a problem. Even if it doesn't indicate a problem,
  // it'd be nice not to call zip with mismatched sequences because
  // it makes it harder to spot the real problems.
```

3 CopyAnalysis

```
TODO
```

3.1 Suggestions to improve performance

The second version below should be faster (it avoids replacing with the same instruction, thus avoiding `touched == true`, thus avoiding DFA iterations):

- Original:

```
case _ =>
  bb.replaceInstruction(i, LOAD_LOCAL(info.getAlias(l)))
  log("replaced " + i + " with " + info.getAlias(l))
```

- Alternative:

```
case _ =>
  val al = info.getAlias(l)
  if(al ne l) {
    bb.replaceInstruction(i, LOAD_LOCAL())
    log("replaced " + i + " with " + info.getAlias(l))
  }
```

3.2 Questions

When computing `copyLattice.lub2()`, two non-bottom stacks `a` and `b` are merged as follows:

```
val resStack =
  if (exceptional) exceptionHandlerStack
  else {
    (a.stack, b.stack).zipped map { (v1, v2) =>
      if (v1 == v2) v1 else Unknown
    }
  }
```

A merged stack slot is taken to be `Unknown` unless both merged values (`v1` and `v2`) are the same. However, although unequal, `v1` and `v2` may still denote the same source (say, `Deref(LocalVar(abc)` and `Deref(Field(r1, f1))` where in turn `getFieldValue(r1, f1) == Deref(LocalVar(abc) )`.

Similary when merging the values of locals (ie, when computing `resBindings`). In that case the comparison reads `v == b.bindings(k)`

Apparently the situation above is possible, because when computing the abstract state, the *source value* isn't looked up. For example:

```
case LOAD_LOCAL(local) =>
  out.stack = Deref(LocalVar(local)) :: out.stack

  /*- there might have been a binding for `local` in the `in` argument
      (i.e., in the pre-instruction abstract state).
      Similarly for LOAD_FIELD */
```

```
TODO

Questions:

 (1) should ``source'' values be added (when available) by interpret()?

 (2) if not, can v1 and v2 be canonicalized before comparing for equality when merging them?
     Although v1 and v2 refer to accesses in different control-flow paths,
     but still their canonicalizations are comparable.
```

3.3 Ideas for the future

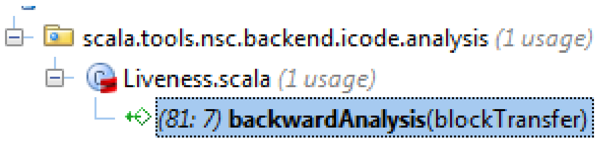
The current definition of "`Deref(LocalVar(l))`" is not program-point-aware, and thus the need to turn *abstract values in the stack* into `Unknown` after a local is assigned:

```
    /** Remove all references to this local variable from both stack
     *  and bindings. It is called when a new assignment destroys
     *  previous copy-relations.
     */
    final def cleanReferencesTo(s: copyLattice.State, target: Location) {
```

Some ideas from points-to analysis could find their way into a more fine-grained (yet efficient) representation for abstract values.

4 LivenessAnalysis

`LivenessAnalysis` is a backward DFA (data-flow analysis) (the only backward-DFA of those in the compiler).



TODO

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. `ClosureElimination` and `DeadCodeElimination`, 2011. Notes at *The Scala Compiler Corner*. `http://lamp.epfl.ch/~magarcia/ScalaCompilerCornerReloaded/2011Q4/ClosureOptimiz.pdf`.

[5] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.