Using reaching-defs and type-flow analyses to obtain three-address code in the Scala compiler

© Miguel Garcia, LAMP,
École Polytechnique Fédérale de Lausanne (EPFL)
http://lamp.epfl.ch/~magarcia

April 24th, 2010

Contents

Abstract

We review first the ICode mechanisms needed for jimplification (previously, we had used Soot for that). Some differences between ICode and JVM bytecode make easier obtaining 3-address form. For example, local slots are typed in ICode, and stack slots are not reused with different types. The ICode mechanisms we employ are: (a) fixpoint on type-stacks and abstract interpretation on type-stacks; and (b) reaching defs for stack slots and local variables. We show how our prototype evolves from a version that only displays results from those analyses, to the final version where ASTs for A3Code are built.

1 What's new over previous write-ups

Previous write-ups provided background information on the conversion from bytecode into 3-address form:

- *Jimple to C# proof-of-concept* `http://www.sts.tu-harburg.de/people/mi.garcia/ScalaCompilerCorner/GenCSharp01.pdf`
- *Towards a typed 3-address IR from ICode* `http://www.sts.tu-harburg.de/people/mi.garcia/ScalaCompilerCorner/StacklessICode.pdf`

Sec. 2 covers jimplification in Soot 2.4.0 (the steps below) in more detail:

1. (stack-based) bytecode is converted into untyped 3-address form.
2. Bellamy's type inference algorithm [1] gives types to local variables.
3. a post-processing step disambiguates bytecode-level usages of `int` into Java-level `boolean`, `byte`, `char`, or `int` [1, § 5]. This step is not necessary for ICode because "small-ints" weren't lumped together in the first place.

In Sec. 3, the beginngs of a compiler plugin are described that converts ICode into 3-address form, which can be used as IR to generate C#. Another hurdle to overcome before obtaining C# are the dollar signs in mangled names, which C# compilers don't like. A re-mangling step is necessary.

2 Jimplification in Soot

2.1 ICode locals are typed, bytecode slots aren't

`ICode` contains instructions not present in JVM bytecode (e.g., `LOAD_MODULE`, Figure 1) and is in fact more expressive. For comparison, the factory methods in `soot.jimple.Jimple` create Jimple CST/AST nodes (the same class hierarchy is used when building Jimple CST trees from parsing, and AST trees from transformations, Figure 2). Also for comparison, the statement hierarchy of Jimple is shown in Figure 3a). The other syntactic category that comes into play is for bytecode and rooted at `soot.coffi.Instruction`.

Regarding the CFGs in Soot and ICode, the following differences exist (not sure yet whether jimplification will thus be different for `ICode`):

- the algorithm described in Sec. 2.3 initially builds a list of possible successors for each instruction, including those resulting from exceptional control flow. In contrast, ICode keeps CFGs consisting of `BasicBlock`s, and no explicit edges reflect exceptional control flow.
- Soot follows the JVM convention where local variables are represented as untyped slots. In common with Soot, ICode has no assignment statement. However, local variables in ICode are typed. This also causes a difference in terminology: the Soot documentation frequently uses the term *local variable* only for formal parameters and for `this` (in non-static methods), reserving the term *untyped variable* for those added during jimplification.

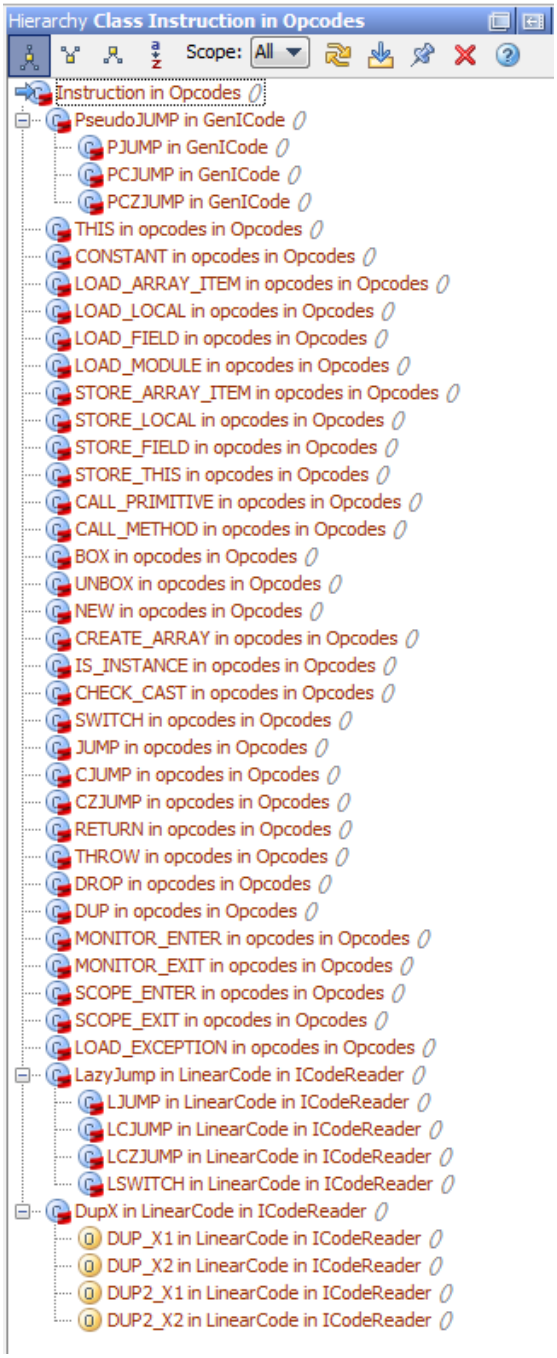

Figure 1: ICode `Instructions`

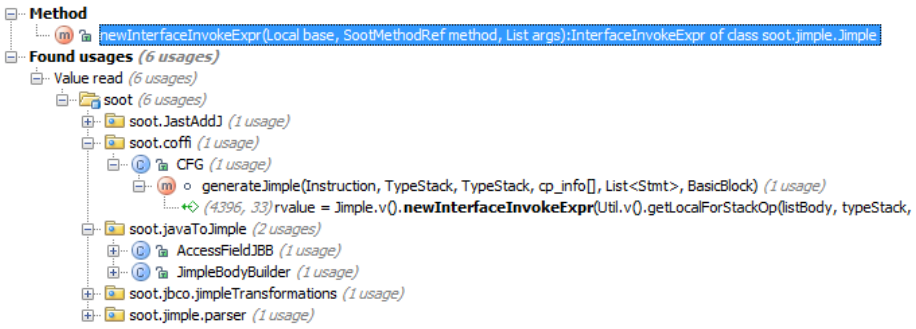

Figure 2: Where factory methods to create Jimple are used

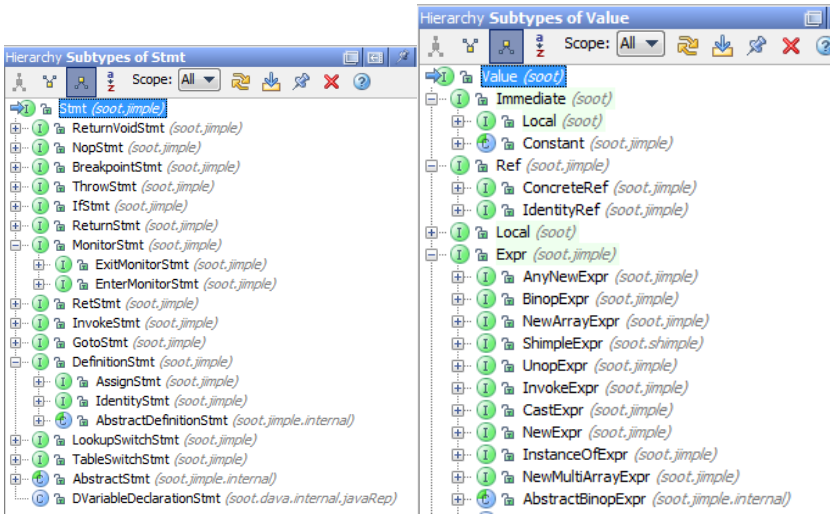

Figure 3: Statement and expression hierarchies of Jimple

Listing 1: An ICode `Local`

```
/** Represent local variables and parameters */
class Local(val sym: Symbol, val kind: TypeKind, val arg: Boolean) {
  var index: Int = -1

  /** Starting PC for this local's visbility range. */
  var start: Int = _

  /** Ending PC for this local's visbility range. */
  var end: Int = _

  /** PC-based ranges for this local variable's visibility */
  var ranges: List[(Int, Int)] = Nil

  override def equals(other: Any): Boolean = (
    other.isInstanceOf[Local] &&
    other.asInstanceOf[Local].sym == this.sym
  );

  override def hashCode = sym.hashCode

  override def toString(): String = sym.toString()
}
```

- Printouts of ICode do not show the type of locals by default, but this information is available in `Local.kind` (Listing 1) and could be printed from `Printers.TextPrinter.printMethod(m: IMethod)` in package `scala.tools.nsc.backend.icode`.

There are also differences in the compilation pipelines leading to bytecode and ICode. The paper by Gagnon *et al.* [2] shows examples where it's necessary to create dedicated variables for independent uses of the same slot. However, these examples (Listing 2 and Listing 3) do not lead to ICode exhibiting untyped local variables. For example, ICode for Listing 3 is shown in Listing 4.

Listing 2: `Harder` from [2], Scala version

```
class Harder extends Object {
  def harder() {
    var y : <untyped> /*- <--- this <untyped> stuff is possibly in JVM bytecode */
                      /*-     but not in Scala and thus not in ICode      */
    if(System.currentTimeMillis() == 0) {
      y = new CA(); y.f();
    } else {
      y = new CB(); y.g();
    }
    y.toString()
  }
}
```

Listing 3: `Hardest` from [2], Scala version

```
class InterfaceDemo {
  def getC() : IC = new CC();
  def getD() : ID = new DD();

  def hardest() {
    var aa = if(System.currentTimeMillis() == 0)
      getC();
    else
      getD();

    aa.f() // invokeinterface IA.f
    aa.g() // invokeinterface IB.g
  }

  class CC extends IC {
    def f() {}
    def g() {}
  }

  class DD extends ID {
    def f() {}
    def g() {}
  }

  trait IA { def f(); }
  trait IB { def g(); }
  trait IC extends IA with IB {}
  trait ID extends IA with IB {}

}
```

2.2 `ICodeReader.parseByteCode()`

As part of `Inliners`, the object `icodereader` is invoked to obtain ICode from bytecode.

```
  object icodeReader extends ICodeReader {
    lazy val global: ICodes.this.global.type = ICodes.this.global
  }
```

Types are ascribed to local variables, but still the resulting ICode makes use of implicit variables (i.e., positions on the evaluation stack) that get no type ascribed. Although ICode allows typed local variables, it has no instruction to directly assign one local to another: those data transfers have to go through the evaluation stack. Similarly, the results of method invocations and of operators applications are left on the stack. Summing up, the only way for a local to be assigned a value is from the top of the stack.

2.3 From stack-based to untyped 3-address

The main reference for this phase is the tech report [4, § 3]. The entry point is `soot.coffi.CFG.jimplification()`. Before that invocation, a control flow graph for the given method is built (when running the constructor invocation `new soot.coffi.CFG(coffiMethod)`).

Listing 4: `Hardest` from [2], ICode version

```
class InterfaceDemo extends java.lang.Object, ScalaObject {

  . . .

  def hardest(): Unit {
  locals: variable aa
  startBlock: 1
  blocks: [1,2,3,4]

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

  2:
    7   THIS
    7   CALL_METHOD InterfaceDemoInterfaceDemo.getC (dynamic)
    7   JUMP 4

  3:
    9   THIS
    9   CALL_METHOD InterfaceDemoInterfaceDemo.getD (dynamic)
    6   JUMP 4

  4:
    6   STORE_LOCAL variable aa
    6   SCOPE_ENTER variable aa
    11  LOAD_LOCAL variable aa
    11  CALL_METHOD InterfaceDemo$IAInterfaceDemo$IA.f (dynamic)
    12  LOAD_LOCAL variable aa
    12  CHECK_CAST REFERENCE(InterfaceDemo$IB)
    12  CALL_METHOD InterfaceDemo$IBInterfaceDemo$IB.g (dynamic)
    12  SCOPE_EXIT variable aa
    12  RETURN (UNIT)

  }
  Exception handlers:

}
```

This step computes `TypeStack`s pre- and post-instruction as well as a pre-instruction array of `TypeLocal`, for the sole purpose of making untyped local variables that approximate the (true) use-defs relationships between stack locations at different program points during execution. Given that ICode already provides us with similar information, we won't need to build these maps. Locals themselves are created in `CFG.generateJimple()` and `getLocalForStackOp`.

Afterwards, some of the (untyped) variables have to be "linked" [4, § 3.4], because in fact they were the same variable. The stack positions to link (called "slots" in [4, § 3.4]) are represented as pairs *(instruction, stack height)*.

Given that the ICode infrastructure contains a `ReachingDefinitions` analysis, we'll rely on it to avoid creating duplicate locals.

2.4 Bellamy's type inference: `TypeResolver`

Code to look at: `soot.jimple.toolkits.typing.fast.TypeResolver`.

The step that need not be performed for ICode (disambiguating small integers) is done in `soot.jimple.toolkits.typing.fast.TypeResolver.typePromotion(Typing)`.

The original Soot algorithm for inserting casts (still necessary in `fast.TypeResolver` in a few cases) is implemented by `soot.jimple.toolkits.typing.integer.TypeResolver.resolve(JimpleBody)`.

3 Prototypes using ICode mechanisms

3.1 First prototype: seeing what ICode knows

Our first prototype will compute nothing itself but display instead the results of two forward dataflow analyses (`MethodTFA` and `ReachingDefinitions`) alongside the ICode instructions (program points) to which they apply. The prototype will thus set the stage for later functionality, allowing first to get some plumbing code right. A brief description of type-flow analysis appears in Ch. 2 of *Towards a typed 3-address IR from ICode*[1].

The prototype is a compiler plugin that adds an ICode-processing phase to run before all ICode optimizations are performed, i.e. right after `GenICode`. Our custom phase:

- gathers the aforementioned information (types for stack positions, and instructions that loaded a value into stack positions), and then
- writes a `.cs` file for each compilation unit, containing all ICode classes for that compilation unit, showing for each instruction the information mentioned above.

The part in charge of writing files to disk is based on *Scala X-Ray* (thanks Mark!) (BTW, a code walkthrough on how Scala X-Ray works appers in Ch. 3 of the *Untangling Scala ASTs* write-up[2]). The part in charge of computing type-stacks and reaching-defs is based on `GenJVM` and `Inliners`. To give a first idea on these steps, Listing 9 shows what our ICode-processing phase looks like at this moment (displaying output on the console for now). This snippet may be simple-minded, but it already allows visualizing (Listing 5) the workings of the ICode utilities (in this case, how a lattice element is updated one ICode instruction at a time, by performing abstract interpretation).

3.2 Second prototype: setting up maps inspired in Soot

The information on pre- and post-instruction type stacks can be obtained with `MethodTFA`, as well as that for reaching definitions (with `ReachingDefinitions`). In Soot, the latter is represented in the map `changedInstructions` (for an `Instruction` loading onto the stack, it returns the instruction(s) reading that stack slot). In the paper by Bellamy *et al.* [1] that same map is called `depends`. That name hints to the fact that, when creating locals from stack slots, use-defs should be

[1] `http://www.sts.tu-harburg.de/people/mi.garcia/ScalaCompilerCorner/StacklessICode.pdf`

[2] `http://www.sts.tu-harburg.de/people/mi.garcia/ScalaCompilerCorner/UntanglingScalaASTs1ofN.pdf`

followed as close as possible, otherwise more instructions than necessary will be flagged as dependent on a stack-slot-loading instruction.

The maps we'll set up in this iteration are `preStack(IMethod, BasicBlock, InstructionIndex)` and `postStack(IMethod, BasicBlock, InstructionIndex)`, which are identical in purpose to the maps used in Soot (`instructionToTypeStack` and `instructionToPostTypeStack`). The remaining map used in Soot, from local slots to type, is not needed in our context because there are no local slots in ICode but typed local variables instead. And their type does not change across instructions as in JVM bytecode. Another Soot map we won't reproduce is `instructionToSuccessors`.

A caveat: whenever "instruction" is mentioned in Soot literature, we use instead a pair (`BasicBlock`, `zeroBasedIndex`) to stand for that program point, because `Instruction`s are `case` classes, and thus given an `Instruction` *value* we can't reliable find a "single" owning block for it. Try yourself:

```
val basicBlock = scala.collection.mutable.Map[Instruction, BasicBlock]()
for (bb <- method.code.blocks) {
  for (ins <- bb.toList) {
    if(basicBlock.isDefinedAt(ins)) {
      scala.Console.println("problem") /*- <- This in general is hit */
    }
    basicBlock += ins -> bb
  }
}
```

The computed reaching definitions give for each stack slot the instruction(s) that have written the slot. As slots are pushed on top, the depth of a slot goes down (obviously but keep reading) which means that the same instruction will appear multiple times as defining slots at different depths for different program points. 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 6 (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 + "\\"
}
```

3.3 AST classes for Stackless ICode

The semantics of ICode and 3-address-ICode are the same. In particular the semantics of object initialization. That semantics dictates how `LOAD_MODULE`, `NEW`, and other ICode instructions are implemented. Well, in 3-address there are (assignment) `FROM_MODULE`, `FROM_NEW`, and so on. Values go to local variables rather than stack slots. That's all. Examples below and in Listing 7.

```
abstract case class ThreeAddressStmt {}

/** 'lhs' gets a value assigned from the element of array 'arr' at index 'index'. */
case class FROM_ARRAY_ITEM(lhs: Local, arr: Local, index: Local) extends ThreeAddressStmt {
  override def toString(): String = lhs+" = "+arr+"["+index+"]"
}

/** 'lhs' gets a value assigned from 'var' */
case class FROM_VAR(lhs: Local, rhs: Local) extends ThreeAddressStmt {
  override def toString(): String = lhs+" = "+rhs
}

case class A3_JUMP(whereto: BasicBlock) extends ThreeAddressStmt {
  override def toString(): String ="goto "+whereto.label
}

/** assigns the value of 'rhs' to 'instance.field' (non-static) or to 'field' (static) */
case class TO_FIELD(instance: Local, field: Symbol,
                    isStatic: Boolean, rhs: Local) extends ThreeAddressStmt {
  override def toString(): String = {
    val fieldName = (if (isStatic) field.fullName else field.toString())
    val instanceNameDot = (if (isStatic) "" else instance+".")
    instanceNameDot + fieldName+" = "+rhs
  }
}
```

3.4 How to run the prototypes

The prototypes can be run in debug mode from IntelliJ IDEA after creating a launch configuration as follows:

```
Main class: scala.tools.nsc.Main
VM parameters: -Xbootclasspath/a:scala-compiler.jar;scala-library.jar;fjbg.jar
Program parameters:
        -Xplugin:Z:\scalaproj\scala\srcmyplugins\a3.jar
        -d Z:\scalaproj\simple\Hardest
        Z:\scalaproj\simple\Hardest\InterfaceDemo.scala
        Z:\scalaproj\simple\Hardest
```

4 A3Code, finally

4.1 Computing webs

A *web* (for a given variable) is the maximal union of intersecting du-chains for the variable. Details in [3, §8.10] (in the context of register allocation, but we need webs instead to determine the *least common ancestor* over all *left-hand side* occurrences of an as-of-yet untyped, about-to-created, local variable).

The data structures supporting the computation of webs are:

```
case class StackAccessingProgramPoint(ppbb: BasicBlock,
                                      ppidx: Int,
                                      ppDepthWithinInstr: Int // 0 for all push instructions
                                     ) {
  override def toString() = { "// in block " + ppbb + ",
    instruction " + ppbb(ppidx) + ", depth " + ppDepthWithinInstr + "\\" }
}

case class UseDefWeb(defs: scala.collection.mutable.Set[StackAccessingProgramPoint],
                     uses: scala.collection.mutable.Set[StackAccessingProgramPoint])
```

Given the above, we'll look up a web using a def or a use as key:

```
  val useDefWebs = new scala.collection.mutable.HashSet[UseDefWeb]() {

    def lookupForDef(bb: BasicBlock, idx: Int) : Option[UseDefWeb] = {
      val candidates = this filter (_ containsDef(bb, idx))
      assert (candidates.size <= 1)
      if(candidates.isEmpty) None
      else Some(candidates.first)
    }

    def lookupForUse(bb: BasicBlock, idx: Int, depth: Int) : Option[UseDefWeb] = {
      val candidates = this filter (_ containsUse(bb, idx, depth))
      assert (candidates.size <= 1)
      if(candidates.isEmpty) None
      else Some(candidates.first)
    }

  }
```

The main steps of the implementation (`computeWebs`) are as follows:

```
    def computeWebs(cls: IClass, method: IMethod) : scala.collection.mutable.Set[UseDefWeb] = {
      rdef.init(method)
      rdef.run
      // use-def webs as in Sec. 8.10 of Muchnick's Advanced Compiler book
      val useDefWebs = /*- as shown above */
      for (currentBlock <- method.code.blocks) {
        var rdefinfo = rdef.in(currentBlock)
        for ((currentInstruction, currentIdx) <- currentBlock.toList.zipWithIndex) {
          if (currentInstruction.consumed > 0) {
            // the info on reaching-defs for the current instruction
            val icodes.reachingDefinitions.rdefLattice.IState(
                 rdefVars, rdefStack : List[Set[(BasicBlock, Int)]]) = rdefinfo
```

All of the above was just to pick an instruction that consumes *something* from the evaluation stack. And in fact, it may consume more than one slot, i.e. for each slot we've got to compute its web.

```
// for each popped value
for (currentPoppedDepth <- 0 to (currentInstruction.consumed - 1) ) {
  val originatingSet = rdefStack(currentPoppedDepth)
```

The above looks up the definitions whose values may flow to the stack slot in question. We call that set of definitions "`originatingSet`," i.e. the set with instructions that may give origin to the slot's value. In terms of Muchnick's book, this is a ud-chain, not a du-chain as explained in his algorithm.

Now we need to find up to one web given those defs and the use. I added some error-checking:

```
  for(originating <- originatingSet) {
    val webIfAny = useDefWebs.lookupForDef(originating._1, originating._2)
    webIfAny match {
      case None => {
        /* the definition not yet in any web, but the use may be.
           If so, add the definitions to that web */
        val webIfAny2 = useDefWebs.lookupForUse(currentBlock, currentIdx, currentPoppedDepth)
        webIfAny2 match {
          case None => {
            /* neither the definition nor the use are in any webs,
               create a new one (with the current use as single use so far)
             */
            useDefWebs += new UseDefWeb(originating._1, originating._2,
                                        currentBlock, currentIdx, currentPoppedDepth)
          }
          case Some(web2) => {
            web2.addDefinitions(originatingSet)
          }
        }
      }
      case Some(web) => {
        /* the definition was in a web, update in place the set of uses
         * in that web adding the current use,
         * and add all other definitions in originatingSet to that web too. */
        web.addUse(currentBlock, currentIdx, currentPoppedDepth)
        if(originatingSet.size > 1) {
          web.addDefinitions(originatingSet)
        }
        /* TODO nothing else to do for currentPoppedDepth, skip the rest of
         * the traversal over originatingSet */
      }
    }
  }
```

Finally, get things ready for the next instruction to check, return the computed webs (as a set) after the last one, and we're out.

```
        }
      }
      /* get the reaching definitions ready for the next instruction in this basic block. */
      rdefinfo = rdef.interpret(currentBlock, currentIdx, rdefinfo)
    }
  }
  useDefWebs
}
}
```

TODO It would be great to have a Scala-based DSL for `.dot` files, rather than helper methods like `useDefWebs2Console(method: IMethod, useDefWebs: collection.Set[UseDefWeb])`. Another option is to install a custom visualizer for the IDEA or Eclipse debuggers.

4.2 Once we have the webs

For each web,

1. create a local variable whose `Type` is merged over those of all uses in that web. These types are given by `preStack`. Should be the same type as merging over all `TypeKind` and then calling `getType` for it (that makes for a nice `assert`).

2. associate the just created local var and the web in question

```
/** Cut and pasted from GenICode.Context
 * Make a fresh local variable. It ensures the 'name' is unique. */
def makeLocal(unit: CompilationUnit, method: IMethod,
              pos: Position, tpe: Type, name: String): Local = {
  val sym = method.symbol.newVariable(pos, unit.fresh.newName(pos, name))
    .setInfo(tpe)
    .setFlag(scala.tools.nsc.symtab.Flags.SYNTHETIC)
  method.addLocal(new Local(sym, toTypeKind(tpe), false))
}
```

For each basic block, for each instruction:

- if the instruction pushes, (a.1) look up the web where it appears as a def; (a.2) the local var in that web will be on the LHS of the new `A3` statement.

- if the instruction pops (this condition and the one above do not exclude each other), (b.1) look up the web where it appears among the uses *with the depth of interest*; (b.2) the *ith* var being read becomes the *jth* operand of the new `A3` statement (i and j may differ because of one or more constants as operands provided by the ICode instruction).

- don't forget to also translate the instructions that neither push nor pop.

The algorithm above creates redundant vars from, for example, `DUP` instructions (which consumes 1 and produces 2 slots, thus finishing a web and starting two). Copy and constant propagation could be performed on the `A3` output. The only optimization of this kind we perform avoids creating another local variable for those webs all of whose definitions are `LOAD_LOCAL`s from the same variable. Given that no assignments to that variable occur in-between uses and defs, we can use the variable directly.

Given that the ICode instructions to be "replaced" were referred from other places (see trait `ExceptionHandler` in Listing 8), those referrers either have to be converted to point to the corresponding `A3` instructions, or the correspondence between ICode and A3 instructions has to be kept somewhere.

Talking about `ExceptionHandler`, when creating its `TryCatch` equivalent in `A3`, remove the jumps at the end of the blocks protected by the clauses in the `TryCatch`.

4.3 The fine print

The previous subsection contains the overview and here we take a look at implementation details (is this refinement? Maybe).

Least common ancestor

The type for a new local variable is computed over those of its uses, by passing a `preTypeStack` to `leastCommonAncestor` in a web:

```
case class UseDefWeb(defs: scala.collection.mutable.Set[StackAccessingProgramPoint],
                     uses: scala.collection.mutable.Set[StackAccessingProgramPoint]) {
  . . .
  /** over all the types of uses in this web */
  def leastCommonAncestor(method: IMethod,
                          preTypeStack: scala.collection.Map[(IMethod, BasicBlock, Int), List[TypeKind]]) = {
    var ts = for (StackAccessingProgramPoint(ppbb, ppidx, ppDepth) <- uses)
             yield (preTypeStack(method, ppbb, ppidx))(ppDepth)
    val t = ts reduceLeft icodes.lub
    t
  }
```

Asymmetry when creating local variables

In the previous section it was mentioned that "[to] avoid creating another local variable for those webs all of whose definitions are `LOAD_LOCAL`s from the same variable. Given that no assignments to that variable occur in-between uses and defs, we can use the variable directly". The core of that decision can be seen in two `UseDefWeb` methods:

```
  def defsAsInstructions
    = defs map { case StackAccessingProgramPoint(ppbb, ppidx, ppDepth) => ppbb(ppidx) }

  def allDefsFromSameVar : Option[Local] = {
    assert (!(defs.isEmpty))
    val b = defsAsInstructions forall (
              ins => ins.isInstanceOf[opcodes.LOAD_LOCAL])
    if(!b) return None
    val loadedVars : scala.collection.Set[Local]
      = for (d <- defsAsInstructions;
             val lolo = d.asInstanceOf[opcodes.LOAD_LOCAL]) yield lolo.local
    if (loadedVars.size == 1) Some(loadedVars.first)
    else None
  }
```

We say "asymmetry" because without this optimization there's a bijection between new locals and webs. With the optimization, there are fewer new locals than webs :-)

Mapping webs to locals (be they old or new)

```
  for(web <- useDefWebs) {
    // val UseDefWeb(defs, uses) = web
    val web2local = new scala.collection.mutable.HashMap[UseDefWeb, Local]
    val local = web.allDefsFromSameVar match {
      case Some(local) => local
```

```
    case None => {
      // create new local
      val tk = web.leastCommonAncestor(method, preTypeStack)
      val newLocal = makeLocal(cls.cunit, method,
                               NoPosition, tk.toType, "stackslot")
      assert (newLocal.kind == tk,
              "passed a Type to makeLocal (although a TypeKind was available) and got another TypeKind")
      newLocal
    }
  }
  web2local += (web -> local)
  . . .
```

TODO agree on a token for the `THIS` local variable, and add it to `web2local` whenever it appears as LHS (a new A3 stmt for that assignment is also needed).

5 TODO

5.1 TODO `A3PrettyPrinter`, `A3Phase`

5.2 Understanding `CALL_METHOD`

```
  case class CALL_METHOD(method: Symbol, style: InvokeStyle) extends Instruction {

    . . .

    override def consumed = method.tpe.paramTypes.length + (
      style match {
        case Dynamic | InvokeDynamic => 1
        case Static(true) => 1 /*- Two different styles of call are Static, but */
        case Static(false) => 0 /*- what's the difference btw Static(true)and Static(false)? */
        case SuperCall(_) => 1
      }
    )
```

As things stand now, the following is printed:

```
def getD(): InterfaceDemo$ID {
locals: []
startBlock: 1
blocks: [1]

1:
  0|  NEW REFERENCE(InterfaceDemo$DD)
              , consumed: 0, consumed types: List()
              , produced: 1, produced types: List() /*- <-- producedTypes needs a touch up */
  1|  DUP
              , consumed: 1, consumed types: List()
              , produced: 2, produced types: List()
  2|  THIS
              , consumed: 0, consumed types: List()
              , produced: 1, produced types: List(REFERENCE(InterfaceDemo))
      /*- according to my math, the typestack at this point is [InterfaceDemo$DD, InterfaceDemo$DD, InterfaceD
  3|  CALL_METHOD InterfaceDemo$DDInterfaceDemo$DD.<init> (static-instance)
         /*- <-- this is an static init, should pop only one value, not 2, or do I miss sthg? */
              , consumed: 2, consumed types: List(REFERENCE(java.lang.Object), REFERENCE(InterfaceDemo))
              , produced: 0, produced types: List()
      /*- if CALL_METHOD consumes two, then the typestack contains [InterfaceDemo$DD] */
```

```
  4|  RETURN (REFERENCE(InterfaceDemo$ID))
             , consumed: 1, consumed types: List()
             , produced: 0, produced types: List()
      /*- but this RETURN produces an InterfaceDemo$ID, is a CHECK_CAST implied? */

}
Exception handlers:
```

References

[1] Ben Bellamy, Pavel Avgustinov, Oege de Moor, and Damien Sereni. Efficient local type inference. In Gregor Kiczales, editor, *OOPSLA'07: 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 2008. `http://progtools.comlab.ox.ac.uk/publications/oopsla08abc`.

[2] Etienne Gagnon, Laurie J. Hendren, and Guillaume Marceau. Efficient inference of static types for Java bytecode. In Jens Palsberg, editor, *SAS*, volume 1824 of *Lecture Notes in Computer Science*, pages 199–219. Springer, 2000. `http://www.sable.mcgill.ca/publications/papers/2000-2/sable-paper-2000-2.ps`.

[3] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[4] Raja Vallee-Rai and Laurie J. Hendren. Jimple: Simplifying Java bytecode for analyses and transformations. Technical report, McGill University, School of Computer Science, Sable Research Group, 1998. `http://www.sable.mcgill.ca/publications/techreports/sable-tr-1998-4.ps`.

Listing 5: Output fragment for one block of one of the methods in the example from Listing 4

```
method: InterfaceDemo.hardest
        on entry to block: 4
               rdef info: IState(Set(),List(Set((2,1), (3,1))))
               tfa  info:
                       entry bindings : Map()
                       entry type stack : [REFERENCE(InterfaceDemo$IA)]
              0|     STORE_LOCAL variable aa
                       bindings: Map(variable aa -> REFERENCE(InterfaceDemo$IA))
                       type stack   : []
                       post rdef: IState(Set((variable aa,4,0)),List())
              1|     SCOPE_ENTER variable aa
                       bindings: Map(variable aa -> REFERENCE(InterfaceDemo$IA))
                       type stack   : []
                       post rdef: IState(Set((variable aa,4,0)),List())
              2|     LOAD_LOCAL variable aa
                       bindings: Map(variable aa -> REFERENCE(InterfaceDemo$IA))
                       type stack   : [REFERENCE(InterfaceDemo$IA)]
                       post rdef: IState(Set((variable aa,4,0)),List(Set((4,2))))
              3|     CALL_METHOD InterfaceDemo$IAInterfaceDemo$IA.f (dynamic)
                       bindings: Map(variable aa -> REFERENCE(InterfaceDemo$IA))
                       type stack   : []
                       post rdef: IState(Set((variable aa,4,0)),List())
              4|     LOAD_LOCAL variable aa
                       bindings: Map(variable aa -> REFERENCE(InterfaceDemo$IA))
                       type stack   : [REFERENCE(InterfaceDemo$IA)]
                       post rdef: IState(Set((variable aa,4,0)),List(Set((4,4))))
              5|     CHECK_CAST REFERENCE(InterfaceDemo$IB)
                       bindings: Map(variable aa -> REFERENCE(InterfaceDemo$IA))
                       type stack   : [REFERENCE(InterfaceDemo$IB)]
                       post rdef: IState(Set((variable aa,4,0)),List(Set((4,5))))
              6|     CALL_METHOD InterfaceDemo$IBInterfaceDemo$IB.g (dynamic)
                       bindings: Map(variable aa -> REFERENCE(InterfaceDemo$IA))
                       type stack   : []
                       post rdef: IState(Set((variable aa,4,0)),List())
              7|     SCOPE_EXIT variable aa
                       bindings: Map(variable aa -> REFERENCE(InterfaceDemo$IA))
                       type stack   : []
                       post rdef: IState(Set((variable aa,4,0)),List())
              8|     RETURN (UNIT)
                       bindings: Map(variable aa -> REFERENCE(InterfaceDemo$IA))
                       type stack   : []
                       post rdef: IState(Set((variable aa,4,0)),List())
```

Listing 6: See Sec. 3.2

```
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 bl
       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 bl
       2|       CJUMP (LONG)EQ ? 2 : 3
                last type stack : []
                no reaching-defs on the empty stack
```

Listing 7: Examples of AST classes for 3-address form

```
/** 'lhs' gets a value assigned from the result of a method invocation (void is returned as UNIT,
 *   will be desugared later in GenJVM or GenMSIL)
 *
 * STYLE: dynamic / static(StaticInstance)
 * Stack: ...:ref:arg1:arg2:...:argn
 *    ->: ...:result
 *
 * STYLE: static(StaticClass)
 * Stack: ...:arg1:arg2:...:argn
 *    ->: ...:result
 *
 */
case class FROM_METHOD_CALL(lhs: Local, method: Symbol,
       style: opcodes.InvokeStyle, args: Array[Local]) extends ThreeAddressStmt {

  override def toString(): String =
    hostClass.fullName+"."+method.fullName+" ("+args.mkString(",")+") /* InvokeStyle: "+style.toString()+" */"

  def hostClass: Symbol = method.owner;
  /** This is specifically for preserving the target native Array type long
   *  enough that clone() can generate the right call.
   */
  var targetTypeKind: TypeKind = UNIT // the default should never be used, so UNIT should fail fast.
  def setTargetTypeKind(tk: TypeKind) = targetTypeKind = tk

  def consumed = method.tpe.paramTypes.length + (
    style match {
      case opcodes.Dynamic | opcodes.InvokeDynamic => 1
      case opcodes.Static(true) => 1
      case opcodes.Static(false) => 0
      case opcodes.SuperCall(_) => 1
    }
  )

  def consumedTypes = {
    val args = method.tpe.paramTypes map toTypeKind
    style match {
      case opcodes.Dynamic | opcodes.Static(true) => AnyRefReference :: args
      case _ => args
    }
  }

  def produced =
    if(toTypeKind(method.tpe.resultType) == UNIT) 0
    else if(method.isConstructor) 0
    else 1

}
```

Listing 8: trait `ExceptionHandlers`

```
trait ExceptionHandlers { self: ICodes =>
  import global.{Symbol, NoSymbol};

  class ExceptionHandler(val method: IMethod, val label: String, val cls: Symbol) {
    private var _startBlock: BasicBlock = _;
    var finalizer: Finalizer = _;

    /** Needed for the MSIL backend. */
    var resultKind: TypeKind = _;

    def setStartBlock(b: BasicBlock) = {
      _startBlock = b;
      b.exceptionHandlerStart = true
    }
    def startBlock = _startBlock;

    /** The list of blocks that are covered by this exception handler */
    var covered: immutable.Set[BasicBlock] = immutable.HashSet.empty[BasicBlock]

    def addCoveredBlock(b: BasicBlock): ExceptionHandler = {
      covered = covered + b
      this
    }

    /** Is 'b' covered by this exception handler? */
    def covers(b: BasicBlock): Boolean = covered(b);

    /** The body of this exception handler. May contain 'dead' blocks (which will not
      * make it into generated code because linearizers may not include them) */
    var blocks: List[BasicBlock] = Nil;

    def addBlock(b: BasicBlock): Unit = blocks = b :: blocks;

    override def toString() = "exh_" + label + "(" + cls.simpleName + ")";

    /** A standard copy constructor */
    def this(other: ExceptionHandler) = {
      this(other.method, other.label, other.cls);
      covered   = other.covered;
      setStartBlock(other.startBlock);
      finalizer = other.finalizer;
    }

    def dup: ExceptionHandler = new ExceptionHandler(this);
  }

  class Finalizer(method: IMethod, label: String) extends ExceptionHandler(method, label, NoSymbol) {
    override def toString() = "finalizer_" + label;

    override def dup: Finalizer = new Finalizer(method, label);
  }

  object NoFinalizer extends Finalizer(null, "<no finalizer>") {
    override def startBlock: BasicBlock = error("NoFinalizer cannot have a start block.");
    override def setStartBlock(b: BasicBlock): Unit = error("NoFinalizer cannot have a start block.");
  }
}
```

Listing 9: A snippet from the first prototype, to show reaching-defs and type-stacks on the console

```
  class GenCSharpPhase(prev: Phase) extends ICodePhase(prev)
  {
    def name = GenCSharpPlugin.PluginName

    override def run {
      if (settings.debug.value) inform("[running phase " + name + " on icode]")
      if (settings.Xdce.value)
        for ((sym, cls) <- icodes.classes ; if inliner.isClosureClass(sym) && !deadCode.liveClosures(sym))
          icodes.classes -= sym

      icodes.classes.values foreach apply

      generateOutput()
    }

    override def apply(cls: IClass) {
      for (method <- cls.methods if method.code != null) {

        val rdef = new icodes.reachingDefinitions.ReachingDefinitionsAnalysis
        rdef.init(method)
        rdef.run

        val tfa = new analysis.MethodTFA
        tfa.init(method)
        tfa.run

        import icodes.opcodes.CALL_METHOD
        import icodes.opcodes.Static

        for (bb <- method.code.blocks) {
          scala.Console.println("method: " + method.toString)

          scala.Console.println("\t on entry to block: " + bb)
          var rdefinfo = rdef.in(bb)
          val icodes.reachingDefinitions.rdefLattice.IState(rdefVars, rdefStack) = rdefinfo
          scala.Console.println("\t\t rdef info: " + rdefinfo)

          var tfainfo = tfa.in(bb)
          val tfa.lattice.IState(tfavars, tfastack) = tfainfo
          scala.Console.println("\t\t tfa info: ")
          scala.Console.println("\t\t\t entry bindings : " + tfavars)
          scala.Console.println("\t\t\t entry type stack : " + tfastack.types.mkString("[", ",", "]"))

          for ((ins, idx) <- bb.toList.zipWithIndex) {
            scala.Console.println("\t\t" + idx + "/\t" + ins)

            tfainfo = tfa.interpret(tfainfo, ins)
            val tfa.lattice.IState(entryBindings, typeStack) = tfainfo
            scala.Console.println("\t\t\t bindings: " + entryBindings)
            scala.Console.println("\t\t\t type stack
: " + typeStack.types.mkString("[", ",", "]"))

            /** Return the reaching definitions corresponding to the point after idx. */
            rdefinfo = rdef.interpret(bb, idx, rdefinfo)
            scala.Console.println("\t\t\t post rdef: " + rdefinfo)
          }

        }

      }
    }
  }
```