# Moving Scala ASTs one step closer to C

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

September 26$^{th}$, 2011

**Abstract**

The current backends (JVM and CLR) emit stack-based OO instructions, taking as input an expression language with possibly complex expressions.

These notes describe a compiler plugin ("`imp`", runnning after `cleanup`) that lowers ASTs into a language close to Three-Address-Form (i.e., similar to Soot Jimple). Given that this IR is a subset of that accepted by the `GenICode` phase, the usual compilation pipeline can be used afterwards. What does "`imp`" output look like? For example, each callsite is rewritten to be preceded by assignments of the receiver and arguments into temporary variables, and so on recursively, thus making evaluation order explicit.

Among others, the resulting C-like IR is suitable for tasks like:

- program verification
- translation into C#, Objective-C, or Android NDK
- no need for "type inference" of operand-stack locations (as performed by bytecode verifiers) because the intermediate variables introduced by `imp` are annotated with their types. This can help with:
  - for JVM, preparing stackmap tables for classfiles $\geq$ 50.0,
  - for MSIL, getting rid of false positive `peverify` errors like "`found System.Object, expected` *<some-more-specific-type>*"

Testing "behavioral conformance" against code emitted without `imp` is straightforward:

- The Scala test suite passes `partest --all` when compiled with `imp` (Sec. 2.2).
- `imp` can participate in bootstrapping: a version of `scala-compiler.jar` compiled from the C-like IR successfully compiles its own sources.
- Similarly for `scalacompiler.exe`.

# Contents


**1 Overview** **3**

1.1 How to build and run . . . . . . . . . . . . . . . . . . . . . . . . . . 3

1.2 Input and output languages . . . . . . . . . . . . . . . . . . . . . . . 3

1.3 End-to-end example . . . . . . . . . . . . . . . . . . . . . . . . . . 4

1.4 Conversion recipe and plugin structure . . . . . . . . . . . . . . . . 4

1.5 Is the “imp IR” truly CFG-based or does it have nested statements (if-then-else)? . . . . . . . . . . . . . . . . . . . . . . . . . . 6

**2 Lessons learnt** **8**

2.1 Idioms for `TypingTransformer`s . . . . . . . . . . . . . . . . . . . . 8

2.2 Breaking transformations into testable subsets . . . . . . . . . . . 8

2.3 Local vars and their tables in `GenJVM` . . . . . . . . . . . . . . . . . 9

2.4 Attempt to enter a try block with non-empty stack . . . . . . . . . 9

2.5 Code size . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

# 1 Overview

## 1.1 How to build and run

Following the standard recipe to package a compiler plugin:

1. compile `Imperating.scala` from `http://lampsvn.epfl.ch/trac/scala/browser/scala-experimental/trunk/imp/src`
2. the resulting classfiles (say, at `myplugins/imp/classes`) are packed with

```
jar -cf imp.jar -C myplugins/imp/classes scala -C myplugins/imp/resources/ .
```

3. where `myplugins/imp/resources` contains the plugin manifest `scalac-plugin.xml`

```
<plugin>
        <name>imp</name>
        <classname>scala.tools.imp.ImpPlugin</classname>
</plugin>
```

Afterwards, `scalac` is run with `-Xplugin:where/to/find/imp.jar` (on Linux) and `-Xplugin where/to/find/imp.jar` (on Windows).

A StackOverflow entry[1] covers how to use compiler plugins when building with Maven.

A note of "*historical interst*": A previous version (`r25163`) required two passes, that was one phase too many. These notes have been updated to reflect the improved transformation.

## 1.2 Input and output languages

It's possible to obtain 3-address from a stack-based language, as demonstrated by two compiler plugins that take ICode as input language:

- transforming into SSA, as part of *Scala on LLVM*[2]
- *Using reaching-defs and type-flow analyses to obtain three-address code in the Scala compiler*[3]

From a reusabiliy perspective however, the earlier a transformation is performed the more phases that stand to benefit from it. Currently both backends accept ICode but future "non-VM backends" might do without `GenICode` altogether.

The `imp` transform acts only on method bodies, breaking up the evaluation of sub-expressions and emitting assignment stmts that are buffered in what stands for "the current evaluation context". The types given to expressions in those assignments were already available (in sub-expressions) and in general the transformation never creates new types nor retypes existing expressions (which simplifies its implementation).

For example, expressions involving short-circuit evaluation (a.k.a. `andAlso`, `orElse`) are rewritten as follows:

[1] `http://stackoverflow.com/questions/4955779/how-do-i-set-the-scala-compiler-to-use-a-plugin-when-i-build-us`

[2] Scala on LLVM by Geoff Reedy, `http://greedy.github.com/scala/`

[3] `http://lamp.epfl.ch/~magarcia/ScalaCompilerCornerReloaded/2010Q2/threeaddress.pdf`

1. make explicit eval order:
   - "`op1 conditional-and op2`"
     is rewritten into
     "`if(op1) { op2 } else { false }`"
   - "`op1 conditional-or op2`"
     is rewritten into
     "`if(op1) { true } else { op2 }`"
2. lowering the resulting `if-then-else` by (a) creating a temp var for the result, (b) transforming recursively the `cond`, (c) transforming the `then` branch to assign to the temp var, (d) same thing for the `else` branch. By now the `if-then-else` is a statement and not an expression anymore.
3. `imp` starts a new evaluation context for each branch of the `if-then-else`, i.e. a new buffer for statements is pushed.
4. The occurrence of the original `if-then-else` will be replaced by another one, whose branches assign to the temp var introduced for the whole `if-then-else` (in case it's "value valued"). In case the `if-then-else` had no result to start with (i.e., it had type `UNIT` or `Nothing`) then no temp var is emitted.

As a sidenote, the snippet in charge of the above is (quoting from `BlocksMaker`):

```
/* ---------- sub-category: short-circuit evaluation ---------- */
 case Apply(fun @ Select(op1, _), List(op2)) if fun.symbol == definitions.Boolean_and =>
   // 'op1 conditional-and op2' is rewritten into 'if(op1) { op2 } else { false }'
   val res = typedWithPos(app.pos) { If(op1, op2, Literal(Constant(false))) }
   transform(res) // transform() needed to turn the If into a stmt, that's where op1, op2 will be transformed.

 case Apply(fun @ Select(op1, _), List(op2)) if fun.symbol == definitions.Boolean_or =>
   // 'op1 conditional-or op2' is rewritten into 'if(op1) { true } else { op2 }'
   val res = typedWithPos(app.pos) { If(op1, Literal(Constant(true)), op2) }
   transform(res) // see comment for Boolean_and
```

It's time for an example.

## 1.3 End-to-end example

The example in Listing 1 complements the discussion in Sec. 1.2 on `if-then-else` rewriting. For that input program, the end result (i.e., the `GenICode` input) is shown in Listing 2.

## 1.4 Conversion recipe and plugin structure

The technique used to track "the current evaluation context" is useful in many AST processing tasks and is covered in some length as part of the description of tail-call elimination [1] (and is not reproduced here).

The structure of `imp` is depicted in Figure 1, with the main visitors highlighted.

In OO-frameworks-speak, a *compiler plugin* is a framework extension, i.e. an interrelated set of types that customize several framework types. Without going into details about what those inter-relationships are:

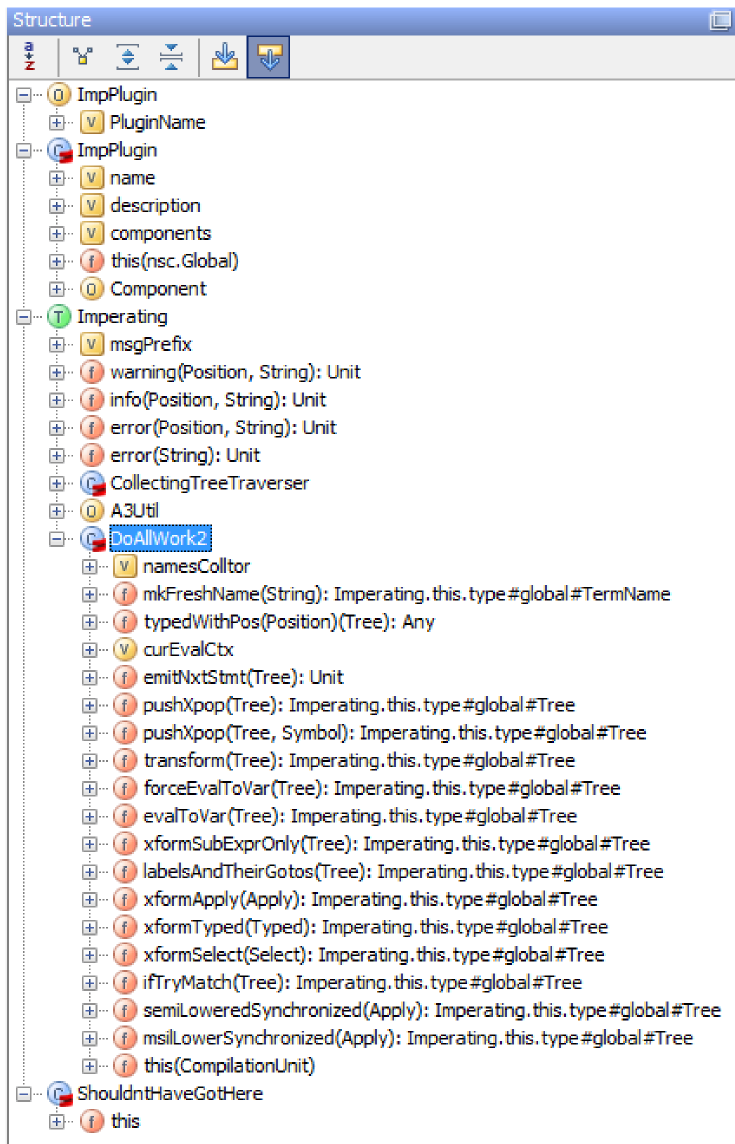


Figure 1: Structure of `imp`, Sec. 1.4

Listing 1: Input program, Sec. 1.3

```
object Test {
  def main(args: Array[String]) {
    val res =
      if(b(1) + b(2) * b(3) > 0) { throw new Exception; "thenBranch" }
      else { "elseBranch" }
  }
  def b(n: Int) = { scala.Console.println(n); n }
}
```

- the plugin class (`scala.tools.imp.ImpPlugin`) extends `Plugin`
- our plugin has a component, which gains its powers from extending the framework-provided `PluginComponent` with a piece of our creation: `trait Imperating`
- that's a hefty trait with the structure depicted in Figure 1)
- however all that complexity remains its own thing, because all `ImpPlugin` needs to know about `Imperating` is how to instantiate its main transformer (appropriately called `DoAllWork2`).

A comments section at the end of `Imperating.scala` includes a visitor for "block flattening" which goes unused (`imp` already emits statement sequences without nested blocks). However it's included because it comes handy to further simplify transformations during prototyping, or to cut down on indentation in log output.

```
// cosmetic unnesting of block-in-block
// (cosmetic because only those Blocks that are easy to remove are removed).
new BlockFlattener transformUnit(unit)
```

The unnesting of the statements from a block directly contained in another block that `BlockFlattener` performs can be seen as "cosmetic" because doing so doesn't mess up evaluation order. In contrast, unnesting the statements from a block contained in a non-block expression requires taking into account the evaluation order for the containing expression, as `imp` does.

### 1.5 Is the "imp IR" truly CFG-based or does it have nested statements (if-then-else)?

`imp` puts the usual Scala AST IR into a straitjacket, albeit without the usual CFG topology, which can be obtained with little additional work. Consider this:

- An "if" in Jimple is

  ```
  if_stmt = "if" condition "then" label;
  ```

- An "if" in `imp` visually looks as nested, e.g.

Listing 2: Sec. 1.3

```
[[syntax trees at end of imp]]// Scala source: bt4.scala
package <empty> {
  final object Test extends java.lang.Object with ScalaObject {
    def main(args: Array[java.lang.String]): Unit = {
      var tmp10: java.lang.String = _;
      val tmp1: Int = Test.this.b(1);
      val tmp2: Int = tmp1;
      val tmp3: Int = Test.this.b(2);
      val tmp4: Int = tmp3;
      val tmp5: Int = Test.this.b(3);
      val tmp6: Int = tmp4.*(tmp5);
      val tmp7: Int = tmp2.+(tmp6);
      val tmp8: Int = tmp7;
      val tmp9: Boolean = tmp8.>(0);
      if (tmp9)
        {
          val tmp11: java.lang.Exception = new java.lang.Exception();
          throw tmp11;
          tmp10 = "thenBranch"
        }
      else
        tmp10 = "elseBranch";
      val res: java.lang.String = tmp10;
      ()
    };
    def b(n: Int): Int = {
      val tmp12: java.lang.Object = scala.Int.box(n);
      scala.Console.println(tmp12);
      n
    };
    def this(): object Test = {
      Test.super.this();
      ()
    }
  }
}
```

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

but in fact can be desugared to ASTs of the form

```
If(Ident(c1) Apply(label)
```

which is the-AST level counterpart to Jimple's "if". Same goes for the other trees in  that involve control flow (`Try`, `Match` – which by now stands for a JVM `tableswitch` or `lookupswitch`).

An excerpt from the Jimple grammar:

```
stmt = breakpoint_stmt | assign_stmt | enter_monitor_stmt |
       goto_stmt | if_stmt | invoke_stmt | lookup_switch_stmt | nop_stmt | ret_stmt |
       return_stmt | return_void_stmt | table_switch_stmt | throw_stmt;
```

# 2 Lessons learnt

## 2.1 Idioms for `TypingTransformers`

The typing of AST nodes we've just built shows up under different idioms, even within a single plugin:

```
typedWithPos(tree.pos) { BLOCK(VAL(tempVar) === EmptyTree, tfedTree, Ident(tempVar)) };
```

or othertimes like this (alternative 2):

```
treeCopy.Apply(app, tfedFun, transformApplyArgs(app))
```

vs. like this:

```
val newIf = If(tfedCond, tfedThenP, tfedElseP)
localTyper typed { newIf }
```

Whenever a `TypingTransformer` is used, the rule of thumb is:

- if a new `Tree` node is created, it should be typed by invoking `localTyper typed` *`newTree`*. This sets the tree's `tpe` (and `symbol` if needed) but not its `pos`, for that a helper `typedWithPos` is the idiom of choice:

  ```
  private def typedWithPos(pos: Position)(tree: Tree) =
    localTyper typed { atPos(pos)( tree ) }
  ```

- say only some children of an existing `Tree` node are to be replaced (those children have a valid `tpe`, somehow). In this case, `treeCopy` can be used, because behind the scenes it invokes `copyAttrs()`:

  ```
  private[scala] def copyAttrs(tree: Tree): this.type = {
    pos = tree.pos
    tpe = tree.tpe
    if (hasSymbol) symbol = tree.symbol
    this
  }
  ```

Additional details on how a `TypingTransformer` manages its `localTyper` can be found in an appendix of the write-up on the `lazyvals` phase [2].

There are also `InfoTransformers`. That's complex. Phases relying on that are shown in Figure 2. *The Scala Compiler Corner* sports write-ups on some of them.

## 2.2 Breaking transformations into testable subsets

As for any semantics-preserving transformation, the Scala testsuite can be enlisted to compare the output of programs compiled with `-Xplugin imp.jar`. This amounts to a small addition in the `partest` script:

```
[ -n "$SCALAC_OPTS" ] || SCALAC_OPTS="-deprecation -Xplugin imp.jar" /*- don't forget the -Xplugin */
```

The above is for Windows. On Linux should read: `-Xplugin:where/to/find/imp.jar`.

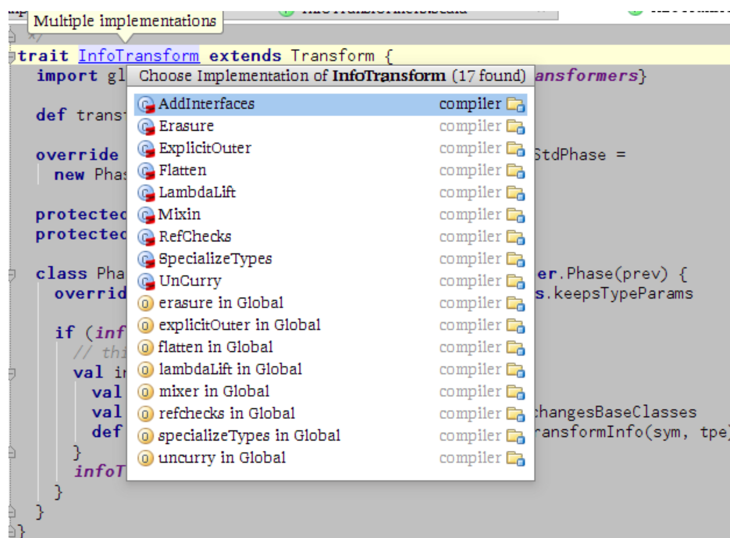


Figure 2: Somewhat off-topic: phases relying on `InfoTransform` (Sec. 2.1)

## 2.3 Local vars and their tables in `GenJVM`

Early in the development of `imp`, compiling a program `forJVM` resulted in

```
java.lang.VerifyError: (class: B$class, method: <clinit> signature: ()V) Illegal local variable number
```

The error refers to a bytecode instruction with an operand (denoting a local var index) that is outside the range of declared local vars. `javap -l` showed such range to be empty for `<clinit>`, because no local vars were emitted by GenJVM for static class initializers (`imp` produces trees where static inits can contain temp vars).

Looking in `genMethod` at how GenJVM emits locals vars (and optionally a table with debug info for them) hints at how to update `addStaticInit` to do the same:

```
for (local <- m.locals if ! m.params.contains(local)) {
  if (settings.debug.value)
    log("add local var: " + local)
  jmethod.addNewLocalVariable(javaType(local.kind), javaName(local.sym))
}

if (emitVars)
  genLocalVariableTable(m, jcode)
```

## 2.4 Attempt to enter a try block with non-empty stack

An earlier version of the Scala.NET compiler emitted assemblies that made `peverify` complain about "*Attempt to enter a try block with non-empty stack*". What follows is the description of a workaround that tells us something about `CleanUp` and `Uncurry`.

Tracking down the bug starts with any of the following peverify errors:

```
[IL] scala.reflect.internal.Symbols+Symbol::isCoDefinedWith][offset 0x000000EB]
     Attempt to enter a try block with nonempty stack.

[IL] scala.reflect.internal.TreePrinters+TreePrinter::printRaw][offset 0x0000144A]
     Attempt to enter a try block with nonempty stack.
```

Next comes pinpointing the Scala expression that resulted in the malformed bytecode, based on the debug info in the `.msil` file (that info includes the original source location).

Another way to find the source location involves compiling with `-Ylog:cleanup` (that's the phase emitting the code pattern where the error shows up).

```
[log cleanup] Dynamically application 'qual1.raw(List())' with
  - declared parameter types: 'List()'
  - passed argument types: 'List()'
  - result type:           '<root>#java#lang#String' - resulting code: {
  val qual1: <root>#java#lang#Object = comment$1;
  try {
  TreePrinters$TreePrinter.this.reflMethod$Method1(qual1.getClass()).invoke(qual1, Array[<root>#java#lang#Object
} catch {
  case (1 @ (_: <root>#java#lang#reflect#InvocationTargetException)) => throw 1.getCause()
}.$asInstanceOf[<root>#java#lang#String]()
}
```

The above information allows placing a breakpoint where cleanup logs that message. Getting to the workaround proper:

```
/*- Before */
case DocDef(comment, definition) =>
  print(comment.raw); println(); print(definition)
```

```
/*- After */
case DocDef(comment, definition) => {
  def getRaw = { comment.raw }

  print(getRaw); println(); print(definition)
}
```

In words, we create a local method whose body just performs the reflective call, to make sure no values have been pushed to the operand stack by the time the reflective call is evaluated. As another example, in `isCoDefinedWith` the expression below contains two reflective calls (`sourceFile.path`). The first invocation finds an empty stack but the second one finds what the first one pushed.

```
  if (this.sourceFile.path != that.sourceFile.path)
    throw InvalidCompanions(this, that)
```

Rather than via a local method we empty the stack by assigning to temp vars:

```
  val thissfp = this.sourceFile.path
  val thatsfp = that.sourceFile.path
  if (thissfp != thatsfp)
    throw InvalidCompanions(this, that)
```

(Somewhat) related to this, `Uncurry` sports a "tree lifting transformation"

(which won't help for the `try-catch` that `cleanup` emits). It works as follows:

```
/** Transform tree `t' to { def f = t; f } where `f' is a fresh name
 */
def liftTree(tree: Tree) = {
  if (settings.debug.value)
    log("lifting tree at: " + (tree.pos))
  val sym = currentOwner.newMethod(tree.pos, unit.freshTermName("liftedTree"))
  sym.setInfo(MethodType(List(), tree.tpe))
  new ChangeOwnerTraverser(currentOwner, sym).traverse(tree)
  localTyper.typedPos(tree.pos)(Block(
    List(DefDef(sym, List(Nil), tree)),
    Apply(Ident(sym), Nil)
  ))
}
```

```
case Try(block, catches, finalizer) =>
  if (needTryLift || shouldBeLiftedAnyway(tree)) transform(liftTree(tree))
  else super.transform(tree)
```

## 2.5 Code size

Needless to say, code is more verbose after `imp` than before, specially because most temp variables are assigned only once (hint: try `-optimize`). This can cause two errors:

- methods in Java classfiles are limited to 65KB, beyond that `GenJVM` throws

  ```
  java.lang.Error: ch.epfl.lamp.fjbg.JCode$$OffsetTooBigException: offset too big to fit in 16 bits: 33190
  ```

- the CLR doesn't have that limitation, but on 32-bit the code section of an assembly compiled with `ilasm /DEBUG` may lead (indirectly) to `StackOverflowException`. Workarounds: either 64-bit or leave out the `/DEBUG` flag.

# References


[1] Miguel Garcia. Rewriting a method body to eliminate recursive tail calls, 2011. *The Scala Compiler Corner*. http://lamp.epfl.ch/~magarcia/ScalaCompilerCornerReloaded/2011Q3/TailCalls.pdf.

[2] Miguel Garcia. What goes on in the `lazyvals` phase, 2011. *The Scala Compiler Corner*. http://lamp.epfl.ch/~magarcia/ScalaCompilerCornerReloaded/2011Q2/HowLazyValWorks.pdf.