

Another backend for the Scala compiler:
GenJava in addition to **GenJVM**

© Miguel Garcia, STS, Hamburg University of Technology
<http://www.sts.tu-harburg.de/people/mi.garcia>

January 12, 2010
Latest version at
[http://www.sts.tu-harburg.de/people/mi.garcia/
ScalaCompilerCorner](http://www.sts.tu-harburg.de/people/mi.garcia/ScalaCompilerCorner)

Contents

1	Motivation and Background	1
1.1	Reasons for a Java backend	1
1.2	Previous work	2
1.3	Generating JavaScript	2
1.4	AST shapes that GenICode expects	2
1.4.1	Nesting of class definitions	2
1.4.2	Kinds of types that have survived so far	2
1.4.3	Statements and expressions reaching this phase	3
2	Translation mechanics	6
2.1	Recovering structured control-flow constructs from local jumps	6
2.2	Pattern matching without forward jumps	7
2.2.1	AST nodes of interest	7
2.2.2	TODO	8
2.3	Bytecode idioms that can't be expressed as-is in Java	8
3	Related Work, Conclusions and Outlook	11
3.1	Related Work	11
3.2	Future Work	11
3.2.1	Porting optimizations from Scala to Java compilers	11
3.2.2	Parallelizing scalac	12
3.2.3	Generating C#	12
3.3	Conclusions	13
4	Useful snippets	16
4.0.1	sending .java files to disk	16
4.0.2	validating resulting Java types down to method signatures	17

Abstract

A previous write-up at *The Scala Compiler Corner*¹ mentioned that the current compiler backend receives ASTs whose typing already allows the resulting bytecode to be accepted by JVMs (for details see *From Scala to Scala--*). With the aim of providing a proof-of-concept, pre-beta version of such translator, these notes summarize a walkthrough of a compiler plugin that generates `.java` files instead of `.class` files.

¹<http://www.sts.tu-harburg.de/people/mi.garcia/ScalaCompilerCorner>

Chapter 1

Motivation and Background

1.1 Reasons for a Java backend

The draft notes on `Scala--` mentioned in passing:

It is thinkable to develop a new backend for the Scala compiler, to generate Java 5 instead of bytecode. The compiler plugin realizing such backend could take as input either `ICode` ASTs (before `ICode`-level optimizations are performed); or `Scala--` ASTs. Admittedly, the ensuing Java compilation would most likely not be competitive with `scalac`, given that Java compilers have not been tuned to cope with the idioms present in `Scala--` (e.g., closure elimination). Alternatively, new optimizations could be added to the bag of tricks of Java compilers.

Coming back to the Scala to Java translator, its main advantages are:

- safety net for decision makers wary of new programming languages.
- Scala in managed environments other than the JVM, e.g. Google's GWT.
- allow reusing program verification techniques (such as JML, <http://www.jmlspecs.org>), Design by Contract, etc.) whose tooling aims at Java. Some JML expressions can be generated from Scala expressions, others from an encoding of JML as an embedded DSL [4].

The last point deserves further explanation. JML tooling has been fighting an uphill battle against the lack of syntax-extensibility that most Java tools exhibit [2] (save for notable exceptions [5]). The annotations about other code fragments required for verification (pre and postconds, for example) can be expressed using Scala syntax, alongside their target code fragments (same applies to loop invariants). Provided one uses only pure expressions in verification contracts, the “by-name parameter” technique (used in Scala's `assert`) can be brought to bear to do away with annotations.

The big advantage in comparison to JML tooling is that the Scala compiler takes over the difficult task of type checking those expressions, and then one needs “only” desugar them into the language of the target verification backend [10].

1.2 Previous work

Regarding previous work, Nikolay Mihaylov¹ and Lex Spoon developed in 2007 a prototype² (`GenJava`) to translate Scala to Java. The approach described in these notes differs however from the architecture of that prototype.

1.3 Generating JavaScript

The best design to address this use case consists in first getting the Scala \rightarrow Java translation right, and then hope for GWT to do its best. The caveats of GWT when translating Java are documented at: <http://code.google.com/webtoolkit/doc/latest/DevGuideCodingBasicsCompatibility.html>

1.4 AST shapes that GenICode expects

The only phases coming after `icode` are `Inliners` (in charge of method inlining), `ClosureElimination` (get rid of uncalled closures), `DeadCodeElimination`, and `GenJVM`. Details at `computePhaseDescriptors()` in `Global.scala`. None of these phases modifies the typing of ASTs significantly, as was done for example by `flatten` (which gets rid of inner classes). In fact, if the compiler option `-optimize` is missing, all three phases (`inline`, `closelim`, and `dce`) are skipped.

1.4.1 Nesting of class definitions

The distinction between `flatten` and `lambdalift` is covered in [1, §3.3]:

In Scala, there are not only local functions but also local classes. This section describes how the lambda lifting technique is generalized to local classes. It is important to keep in mind that lambda lifting aims only at eliminating local definitions, not inner classes, which remain in place. In fact, lambda lifting preserves the nesting of classes; if a class is nested within some class it will still be after lambda lifting. Thus references to enclosing instances are unaffected by lambda lifting

...

Inner classes are not lifted out of their enclosing class. They remain in place but possible references to free variables contribute to the extra fields of their enclosing class and are replaced with references to these fields

1.4.2 Kinds of types that have survived so far

Of all the type representations used at one point or another during compilation (Figure 1.1), only those shown in Listing 1.1 need to be handled by `GenICode` (auxiliary definitions are brought together in Listing 1.2).

¹<http://lamp.epfl.ch/~mihaylov/>

²<http://lampsvn.epfl.ch/trac/scala/browser/scala/branches/jvm-src/src/compiler/scala/tools/nsc/backend/jvm/GenJava.scala>

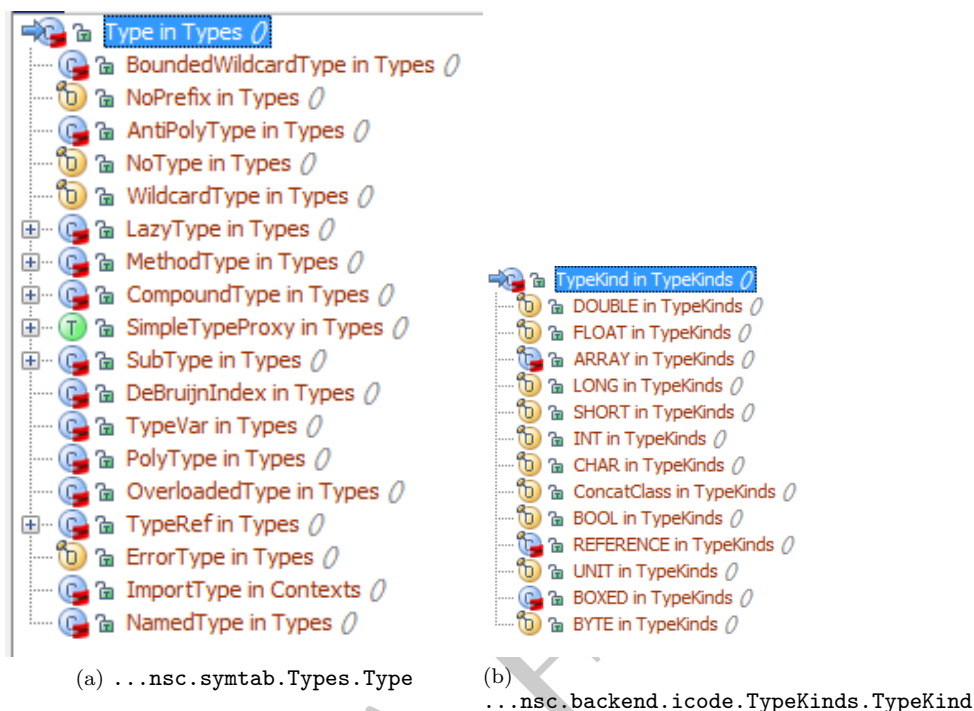


Figure 1.1: Types on the way from Scala to Java

The reasoning goes as follows: if at this stage we can cook a Java type for each type representative returned by `toTypeKind()`, then we've got already half of the translation. Translating expressions and statements constitutes the other half. The only suspicious names in Figure 1.1b are `ConcatClass`, `REFERENCE`, and `BOXED`. The comment for `ConcatClass` reads “*Dummy TypeKind to represent the ConcatClass in a platform-independent way. For JVM it would have been a REFERENCE to ‘StringBuffer’.*”. For `BOXED` it reads: “*A boxed value*”.

1.4.3 Statements and expressions reaching this phase

Before getting to them, the entry points are (from `GenICode.scala`):

```
def gen(tree: Tree, ctx: Context): Context = tree match {
  case EmptyTree => ctx
  case PackageDef(pid, stats) => ...
  case ClassDef(mods, name, _, impl) => ...
  case ModuleDef(mods, name, impl) =>
    abort("Modules should not reach backend!")
  case ValDef(mods, name, tpt, rhs) => ...
  case DefDef(mods, name, tparams, vparamss, tpt, rhs) => ...
  case Template(_, _, body) => gen(body, ctx)
  case _ => abort("Illegal tree in gen: " + tree)
}
```

A Java translator needs to handle those cases, too. No surprises here :-)

As to expressions, they are translated by `genLoad()`, which generates instructions from trees that produce values on the stack:

Listing 1.1: From Scala to JVM type, `TypeKinds.scala`

```
/** Return the TypeKind of the given type */
def toTypeKind(t: Type): TypeKind = t match {

  case ThisType(sym) => if (sym == ArrayClass) AnyRefReference
                        else REFERENCE(sym)

  case SingleType(pre, sym) => primitiveTypeMap.getOrElse(sym, REFERENCE(sym))
  case ConstantType(value) => toTypeKind(t.underlying)

  case TypeRef(_, sym, args) =>
    primitiveTypeMap.getOrElse(sym, arrayOrClassType(sym, args))

  case ClassInfoType(_, _, sym) =>
    primitiveTypeMap get sym match {
      case Some(k) => k
      case None    =>
        if (sym == ArrayClass) abort("ClassInfoType to ArrayClass!")
        else REFERENCE(sym)
    }

  case ExistentialType(tparams, t) => toTypeKind(t)
  case AnnotatedType(_, t, _)    => toTypeKind(t)

  //case WildcardType => // bq: useful hack when wildcard types come here
  // REFERENCE(definitions.ObjectClass)

  case _ => abort("Unknown type: " + t + ", " + t.normalize +
    "[" + t.getClass + ", " + t.normalize.getClass + "]" + " TypeRef? " +
    t.isInstanceOf[TypeRef] + ", " + t.normalize.isInstanceOf[TypeRef])
}
```

```
def genLoad(tree: Tree, ctx: Context, expectedType: TypeKind): Context
```

Instead, `genStat()` handles trees containing statements that do not push a value to the stack. More specialized `gen...` methods are shown in Figure 1.2.

Listing 1.2: Definitions referred to in the code of Listing 1.1

```

lazy val AnyRefReference: TypeKind = REFERENCE(global.definitions.ObjectClass)

/** A map from scala primitive Types to ICode TypeKinds */
lazy val primitiveTypeMap: collection.Map[Symbol, TypeKind] = {
  import definitions._
  collection.Map(
    UnitClass    -> UNIT, BooleanClass -> BOOL, CharClass    -> CHAR,
    ByteClass    -> BYTE, ShortClass  -> SHORT, IntClass     -> INT,
    LongClass    -> LONG, FloatClass  -> FLOAT, DoubleClass  -> DOUBLE
  )
}

/** Reverse map for toType */
private lazy val reversePrimitiveMap: collection.Map[TypeKind, Symbol] =
  collection.Map(primitiveTypeMap.toList map (_.swap) : _*)

/** Return the type kind of a class, possibly an array type. */
private def arrayOrClassType(sym: Symbol, targs: List[Type]): TypeKind = {
  if (sym == ArrayClass) ARRAY(toTypeKind(targs.head))
  else if (sym.isClass) REFERENCE(sym)
  else { assert(sym.isType, sym) // it must be compiling Array[a]
        AnyRefReference }
}

```

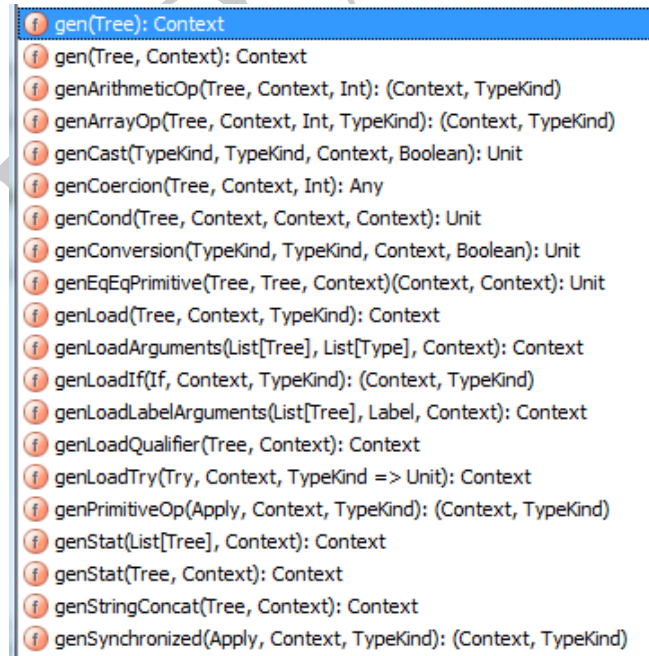


Figure 1.2: Some methods that generate ICode instructions

Chapter 2

Translation mechanics

2.1 Recovering structured control-flow constructs from local jumps

while, do-while, and while(true) loops, as well as forward intraprocedural jumps, are encoded in Scala ASTs in terms of `LabelDef` and `Apply`. In order to generate readable Java syntax (no `gotos`) those nodes can be processed using techniques from [7].

To start, the AST shapes for structured loops are “documented” in Listing 2.1 on p. 8. There’s also a naming convention that both of the parser and `GenICode` abide by (Figure 2.1).

The full `GenICode` snippet reads:

```
def isLoopHeaderLabel(name: Name): Boolean =
  name.startsWith("while$") || name.startsWith("doWhile$")

// used to mark a BasicBlock as being a loop header:
. . .
if (isLoopHeaderLabel(name))
  ctx1.bb.loopHeader = true;
```

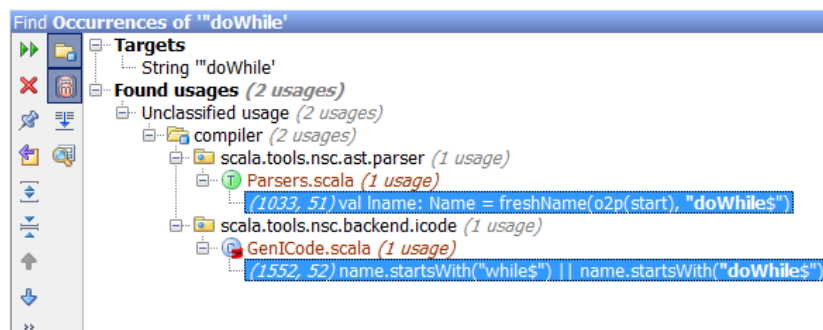


Figure 2.1: An in-band way to detect user-provided loppss

```
// the loopHeader attribute does not affect bytecode generation however,  
// it's used only in backend.icode.Printers
```

Summing up, an `Apply` stands for a jump (as opposed to its “usual” meaning, function application) provided that

```
case app @ Apply(fun, args) =>  
  val sym = fun.symbol  
  if (sym.isLabel) . . . // jump to a label
```

In contrast, a `LabelDef` always denotes the target of a jump. `LabelDef` nodes are added by the transformation for pattern matching (performed during `...ExplicitOuter`) and during `TailCalls`. Later phases do not touch them. Therefore one needs to go all the way to `ICode` to find code examples where `LabelDefs` are processed. Listing 2.2 on p. 9 shows (a) how the parameters to a jump target are used to generate declarations for local variables; and (b) how a map is built to pair the `Tree`-level representation of a jump target with that at the `ICode`-level. The method shown, `scanForLabels`, is invoked upon visiting a forward jump whose target hasn't been encountered so far.

TODO Notice that in most jumps, it is always the case that the `Apply`'s function's symbol matches that of the target `LabelDef`. For `LabelDef-Apply` pairs built by `makeWhile` and `makeDoWhile` this is not the case, as the code in Listing 2.1 shows (nowhere after creating a `LabelDef` is its symbol set to that of the `Ident(name)` of the just created `Apply`).

2.2 Pattern matching without forward jumps

Provided that the only occurrences of `LabelDef-Apply` pairs are due to (a) loops and (b) tail self-calls; the AST can be printed using unmodified Scala textual syntax. Alternatively, the generation of loops for tail self-calls can be skipped altogether with the compiler option `-Yskip:tailcalls` (I also tried `-g:notailcalls` but that did not prevent the `tailcalls` phase from running.)

The difference between `while` and `do-while` loops on the one hand, and loops resulting from tail-calls on the other is that the latter usually declare parameters in the `LabelDef`.

Rather than “un-doing” the transformation (for pattern matching) that generates forward jumps, we explore in this subsection making it generate `if-then-else` tree nodes only.

There are two compiler options related to pattern matching:

```
val Xsqueeze      = ChoiceSetting  
  ("-Ysqueeze", "if on, creates compact code in matching", List("on","off"), "on") .  
  withHelpSyntax("-Ysqueeze:<enabled>")  
  
val Ypmatdebug    = BooleanSetting  
  ("-Ypmat-debug", "Trace all pattern matcher activity.")
```

2.2.1 AST nodes of interest

The AST nodes of interest for pattern matching are shown in Listing 2.3.

Listing 2.1: LabelDef and Apply pairs generated during parsing

```
// The symbol of the LabelDef should have a MethodType (as if a nested function).
// Also in LabelDef, the Idents in params represent parameters.
// Jumps are Apply nodes attributed with the symbol of the target LabelDef,
// the arguments (of the Apply) will get assigned to the idents (of the LabelDef).
// Forward jumps are allowed (the corresponding LabelDef hasn't been visited yet).

case class LabelDef(name: Name, params: List[Ident], rhs: Tree)
  extends DefTree with TermTree {
  assert(rhs.isTerm)
}

// from Trees.scala, "A standard pattern match"
case LabelDef(name, params, rhs) =>
// while/do are desugared to label defs as follows:
// while (cond) body ==> LabelDef($L, List(), if (cond) { body; L$() } else ())
// do body while (cond) ==> LabelDef($L, List(), body; if (cond) L$() else ())

// from TreeBuilder.scala
/** Create tree representing a while loop */
def makeWhile(lname: Name, cond: Tree, body: Tree): Tree = {
  val continu = atPos(o2p(body.pos.endOrPoint)) { Apply(Ident(lname), Nil) }
  val rhs = If(cond, Block(List(body), continu), Literal(()))
  LabelDef(lname, Nil, rhs)
}
/** Create tree representing a do-while loop */
def makeDoWhile(lname: Name, body: Tree, cond: Tree): Tree = {
  val continu = Apply(Ident(lname), Nil)
  val rhs = Block(List(body), If(cond, continu, Literal(())))
  LabelDef(lname, Nil, rhs)
}
```

2.2.2 TODO

TODO It's not clear at this hour whether (a) the current version of pattern matching should be taken as starting point, or (b) an older version (i.e., old enough that forward jumps were not generated).

Sample "old" version:

- <http://lampsyn.epfl.ch/trac/scala/browser/scala/trunk/src/compiler/scala/tools/nsc/transform/ExplicitOuter.scala?rev=17916>
- <http://lampsyn.epfl.ch/trac/scala/browser/scala/trunk/src/compiler/scala/tools/nsc/matching?rev=17916>

Temporary workaround: DON'T GENERATE JAVA FOR SCALA PROGRAMS CONTAINING PATTERN MATCHING, OR ELSE YOU'LL GET AN ERROR MESSAGE :-)

2.3 Bytecode idioms that can't be expressed as-is in Java

- blocks as part of expressions that are executed with non-empty stacks.

Listing 2.2: Handling jumps during ICode generation

```
/**
 * Traverse the tree and store label stubs in the context. This is
 * necessary to handle forward jumps, because at a label application
 * with arguments, the symbols of the corresponding LabelDef parameters
 * are not yet known.
 *
 * Since it is expensive to traverse each method twice, this method is called
 * only when forward jumps really happen, and then it re-traverses the whole
 * method, scanning for LabelDefs.
 *
 * TODO: restrict the scanning to smaller subtrees than the whole method.
 * It is sufficient to scan the trees of the innermost enclosing block.
 */
private def scanForLabels(tree: Tree, ctx: Context): Unit =
  new Traverser() {
    override def traverse(tree: Tree): Unit = tree match {

      case LabelDef(name, params, rhs) =>
        if (!ctx.labels.contains(tree.symbol)) {
          ctx.labels += (tree.symbol
            ->
              (new Label(tree.symbol) setParams(params map (_.symbol)))));
          ctx.method.addLocals(params
            map
              (p => new Local(p.symbol, toTypeKind(p.symbol.info), false)));
        }
        super.traverse(rhs)

      case _ =>
        super.traverse(tree)
    }
  }
  traverse(tree);
```

- object initialization, where outer pointers have to be assigned before calling the superclass
- Overrides in parametric classes require sometimes bridge methods which can lead to double definitions when seen as Java source.

TODO

Listing 2.3: AST nodes of interest for pattern matching

```
/** before TransMatch: Pattern matching expression
 * after TransMatch : Switch statements
 *
 * After TransMatch, cases will satisfy the following constraints:
 * - all guards are EmptyTree,
 * - all patterns will be either Literal(Constant(x:Int)) or Alternative(lit/.../lit)
 * except for an "otherwise" branch, which has pattern Ident(nme.WILDCARD)
 */
case class Match(selector: Tree, cases: List[CaseDef])
  extends TermTree

/** Case clause in a pattern match, eliminated by TransMatch
 * (except for occurrences in switch statements)
 */
case class CaseDef(pat: Tree, guard: Tree, body: Tree)
  extends Tree
```

Chapter 3

Related Work, Conclusions and Outlook

3.1 Related Work

TODO Any generators of C# from high-level languages (for example, does the F# compiler generate bytecode, or does it internally invoke the C# compiler? :-)

3.2 Future Work

3.2.1 Porting optimizations from Scala to Java compilers

The Scala compiler (specially its backend) performs optimizations aimed at exploiting Scala idioms. As far as we know, no detailed cross-check has been performed against recent optimizations that the JVM and Java compilers support. An initial comparison is offered in this section.

Tail Call Optimization

Tail call optimization is making its way to JDK 7 [8], long after a comparable optimization¹ was added to `scalac`. Surprisingly, there's no single-stop source of information on whether mainstream Java compilers support that optimization:

- <http://lambda-the-ultimate.org/node/717>
- http://blogs.azulsystems.com/cliff/2007/05/tailcall_optimi.html
- A patent! <http://www.freepatentsonline.com/y2009/0187884.html>

Closure Elimination, Lambda reification

Rather than offering an up-to-date checklist of optimizations across Scala and Java compilers (listing whether support is overlapping, exclusive to some com-

¹<http://www.sts.tu-harburg.de/people/mi.garcia/ScalaCompilerCorner/TailSelfCallsReport.pdf>

piler, or missing in all compilers), this section just scratches the surface by providing handy links for the reader to peruse further :-)

On the topic of closure elimination, the Scala version is reported in [3] while related work from the Java perspective comprises *object fusion* [9], *stack-allocated objects*, and *lambda reification*:

1. <http://www.artima.com/weblogs/viewpost.jsp?thread=278567>
2. <http://openjdk.java.net/projects/lambda/>
3. javac.info

All these techniques share the observation that the involved objects don't rely on the full `java.lang.Object` contract: e.g., they are never compared for reference equality, and thus the code and references they hold can be held by their (single) owner.

Finding out which optimizations are done (best?) by which compiler (Scala, Java, server-mode JIT) (and which aren't :-)) cannot be gleaned easily from the *current* documentation of the compilers. This analysis is however necessary to exploit synergies and avoid duplication of effort when improving the Scala compiler.

3.2.2 Parallelizing scalac

In addition to the translator into Java, another interesting project involves re-architecting some portions of `scalac` for task-level parallelism, as supported by `java.util.concurrent` [6].

Most of the transformations that each of `superaccessors` till `cleanup` perform, one after the other, one compilation unit after the other, actually operate on their own portion of an AST. For example, the effects of a single tailcalls-rewriting are limited to a single method body. In principle, all those rewritings could be performed in parallel (the other method definitions need not know how a particular method is rewritten). I'm not saying the current implementation of `tailcalls` is thread-safe. What I'm saying is that if compilation phases were structured following the architecture for task-level parallelism then most worries about compilation speed would vanish.

Summing up: all collected ASTs constitute the shared mutable state, each transformation is realized by a component spawning tasks, "component" meaning keeping all of the mutable state required for its operation stack-local or thread-confined.

The programming techniques above (that have to be manually applied) are adopted as the native programming model by X10 and Fortress (trees instead of lists, semi-balanced trees, hierarchical places) only that for numerical computing. Details appear in the talk by Guy Steele, Jr. "*foldl and foldr considered slightly harmful*"². The lessons are the same for AST processing.

3.2.3 Generating C#

Breaking up the translation `Scala--` \rightarrow `Java` into two phases simplifies the development of a backend targeting `C#` source code:

²<http://research.sun.com/projects/plrg/Publications/ICFPAugust2009Steele.pdf>

- in a first step, the incoming ASTs are transformed leaving only constructs that a Java CST (Concrete Syntax Tree) can express directly. Rather than trying to untangle on-the-fly. After this step, ASTs should be “boring” (e.g., no arbitrary jumps, those `Block(stats,expr)` that return void have been sorted from those that don’t, etc.) For completeness, the elimination of forward jumps can be done in different ways:
 - un-applying some previous transformations (e.g., pattern match); or
 - adding new node types (e.g. ”While”, which used to be in the parse tree); or
 - letting jumps be and go for techniques heralded by the Soot Dava decompiler [7]
- the second phase does serialization proper of (.java or .cs) files. Much like `writeICode` does now. Naturally, the incoming AST should include only language constructs supported (with the same semantics) by both of Java and C#. Differences between these languages are catalogued at:
 - http://en.wikipedia.org/wiki/Comparison_of_Java_and_C_Sharp
 - <http://www.geeks.ltd.uk/Knowledgebase/Compare-c-sharp-java.html>

An alternative design to all the above would rely on a yet to be defined intermediate language closer to bytecode than to Java (i.e., in the category of `ICode` but also covering concurrency, for example). Such languages have been proposed, but haven’t gained acceptance so far:

- Intermediate Language Design of High-level Language Virtual Machines. 3rd workshop on Virtual Machines and Intermediate Languages. <http://www.cs.iastate.edu/~design/vmil/2009/papers/p08-marr.pdf>

3.3 Conclusions

TODO

Bibliography

- [1] Philippe Altherr. *A Typed Intermediate Language and Algorithms for Compiling Scala by Successive Rewritings*. PhD thesis, EPFL Lausanne, Switzerland, 2006.
- [2] Austin Clements. A comparison of designs for extensible and extension-oriented compilers. Master’s thesis, Massachusetts Institute of Technology, Feb 2008. <http://pdos.csail.mit.edu/xoc/clements-thesis.pdf>.
- [3] Iulian Dragos. Optimizing Higher-Order Functions in Scala. In *Third International Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, 2008.
- [4] Gilles Dubochet. On Embedding Domain-specific Languages with User-friendly Syntax. In *1st Workshop on Domain-Specific Program Development*, pages 19–22, 2006. <http://infoscience.epfl.ch/record/85862>.
- [5] Torbjörn Ekman and Görel Hedin. The JastAdd extensible Java compiler. In *OOPSLA ’07: Proc. of the 22nd ACM SIGPLAN Conf. on Object Oriented Programming Systems and Applications*, pages 1–18, New York, NY, USA, 2007. ACM.
- [6] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes. *Java Concurrency in Practice*. Addison-Wesley Professional, 2005.
- [7] Jerome Miecznikowski and Laurie J. Hendren. Decompiling Java bytecode: Problems, traps and pitfalls. In *CC’02: Proceedings of the 11th International Conference on Compiler Construction*, pages 111–127, London, UK, 2002. Springer-Verlag. <http://www.sable.mcgill.ca/publications/papers/2002-2/sable-paper-2002-2.ps.gz>.
- [8] Arnold Schwaighofer. Tail call optimization in the java hotspot(tm) vm. Master’s thesis, Johannes Kepler Universität Linz, 2009. <http://www.ssw.uni-linz.ac.at/Research/Papers/Schwaighofer09Master/>.
- [9] Christian Wimmer and Hanspeter Mössenböck. Automatic feedback-directed object inlining in the Java HotSpottm virtual machine. In Chandra Krantz, Steven Hand, and David Tarditi, editors, *VEE*, pages 12–21. ACM, 2007.
- [10] Valentin Wüstholtz. Encoding Scala programs for the Boogie verifier. Master’s thesis, Chair of Programming Methodology, Department of Computer

Science, ETH Zurich, 2009. http://www.pm.inf.ethz.ch/education/theses/student_docs/Valentin_Wuestholz/MA_report.pdf.

DRAFT

Chapter 4

Useful snippets

4.0.1 sending .java files to disk

```
// from Global.scala

/** Returns the file with the given suffix for the given class. Used for icode writing. */
def getFile(clazz: Symbol, suffix: String): File = {
  val outdirname = settings.outputDirs.outputDirFor(clazz.sourceFile)
  var outdir = new File(if (outdirname.path == "") "." else outdirname.path)
  val filename = clazz.fullNameString('.')
  var start = 0
  var end = filename.indexOf('.', start)
  while (end >= start) {
    outdir = new File(outdir, filename.substring(start, end))
    if (!outdir.exists()) outdir.mkdir()
    start = end + 1
    end = filename.indexOf('.', start)
  }
  new File(outdir, filename.substring(start) + suffix)
}

private def writeICode() {
  val printer = new icodes.TextPrinter(null, icodes.linearizer)
  icodes.classes.valuesIterator.foreach((cls) => {
    val suffix = if (cls.symbol hasFlag Flags.MODULE) ".$icode" else ".icode"
    val file = getFile(cls.symbol, suffix)
    //   if (file.exists())
    //     file = new File(file.getParentFile(), file.getName() + "1")
    try {
      val stream = new FileOutputStream(file)
      printer.setWriter(new PrintWriter(stream, true))
      printer.printClass(cls)
      informProgress("wrote " + file)
    } catch {
      case ex: IOException =>
        if (settings.debug.value) ex.printStackTrace()
        error("could not write file " + file)
    }
  })
}
```

4.0.2 validating resulting Java types down to method signatures

```
// from Printers.scala
def printClass(cls: IClass) {
  this.clazz = cls;
  print(cls.symbol.toString()); print(" extends ");
  printList(cls.symbol.info.parents, ", ");
  indent; println(" {");
  println("// fields:");
  cls.fields.foreach(printField); println;
  println("// methods");
  cls.methods.foreach(printMethod);
  undent; println;
  println("}")
}

def printField(f: IField) {
  print(f.symbol.keyString); print(" ");
  print(f.symbol.nameString); print(": ");
  println(f.symbol.info.toString());
}

def printMethod(m: IMethod) {
  print("def "); print(m.symbol.name);
  print("("); printList(printParam(m.params, ", "); print(")");
  print(": "); print(m.symbol.info.resultType)

  if (!m.isDeferred) {
    println(" {")
    println("locals: " + m.locals.mkString(", ", ", "))
    println("startBlock: " + m.code.startBlock)
    println("blocks: " + m.code.blocks.mkString("[", ", ", "]"))
    println
    lin.linearize(m) foreach printBlock
    println("}")

    indent; println("Exception handlers: ")
    m.exh foreach printExceptionHandler

    undent; println
  } else
    println
}

def printParam(p: Local) {
  print(p.sym.name); print(": "); print(p.sym.info);
  print(" ("); print(p.kind); print(")")
}
```
