Scalify

Java -> Scala Source Translator

Target: 100% of Java 1.5

Status: 90% of Java 1.4

Ninety/Ninety rule may apply

http://github.com/paulp/scalify

(BSD-like do-what-you-want license)

Motivations

- Mixed Java/Scala not yet a smooth ride
- Interoperation with Java a many-edged sword
- Education for aspiring Scala programmers
- Lowering barriers to Scala adoption
- Experimenting with exciting and exotic translations: eliminating mutable state, isolating effects, selectively introducing laziness ... there are truckloads of research.
- Actual, less-reverse-engineered reason: fun

Relevance to You

- Yes, it's called "Scalify" but don't let the name fool you
- Ambition is a fully generic frontend and pluggable language-specific backends
- This is more likely to come to pass if someone using another nutty language gets involved before it's too far along

Backend

- All you'll have to do is implement one function!
- "...in 27 parts."
- Maybe more like 95 parts.

- def emit(cu: dom.CompilationUnit): String

Eclipse JDT

- Much of the front end work done for us
- Support for refactoring, searching, compiler warnings, etc. etc.
- Have to run inside OSGI
- ASTParser.createASTs produces one java syntax tree per compilation unit

ASTNode subclasses:

AbstractTypeDeclaration Annotation AnnotationTypeDeclaration AnnotationTypeMemberDeclaration AnonymousClassDeclaration ArrayAccess ArrayCreation ArrayInitializer ArrayType AssertStatement Assignment Block BlockComment BodyDeclaration BooleanLiteral BreakStatement CastExpression CatchClause CharacterLiteral ClassInstanceCreation Comment CompilationUnit ConditionalExpression ConstructorInvocation ContinueStatement DoStatement EmptyStatement EnhancedForStatement EnumConstantDeclaration EnumDeclaration Expression ExpressionStatement FieldAccess FieldDeclaration ForStatement IfStatement ImportDeclaration InfixExpression Initializer InstanceofExpression Javadoc LabeledStatement LineComment MarkerAnnotation MemberRef MemberValuePair MethodDeclaration MethodInvocation MethodRef MethodRefParameter Modifier Name NormalAnnotation NullLiteral NumberLiteral PackageDeclaration ParameterizedType ParenthesizedExpression PostfixExpression PrefixExpression PrimitiveType QualifiedName QualifiedType ReturnStatement SimpleName SimpleType SingleMemberAnnotation SingleVariableDeclaration Statement StringLiteral SuperConstructorInvocation SuperFieldAccess SuperMethodInvocation SwitchCase SwitchStatement SynchronizedStatement TagElement TextElement ThisExpression ThrowStatement TryStatement Type TypeDeclaration TypeDeclarationStatement TypeLiteral TypeParameter VariableDeclaration VariableDeclarationExpression VariableDeclarationFragment VariableDeclarationStatement WhileStatement WildcardType

Eclipse is all Java 1.4, which means

- no generics,
- plenty of raw types
- both failures and missing values typically denoted by null

This aggression will not stand!

```scala
// Scala niftiness #1: Importing a Package
import org.eclipse.jdt.core.dom

object JVMSummit {
  // Scala niftiness #2: Implicit conversions
  // java.util.List[?] => scala.List[T]
  implicit def fixRawLists[T](jlist: java.util.List[_]): List[T] =
    jlist.toArray.toList.map(_.asInstanceOf[T])

  // Scala niftiness #3: a better way than null
  // we can defeat nullness in our lifetimes
  def Opt[T](x: T): Option[T] = if (x == null) None else Some(x)

  // Scala niftiness #4: extractor objects
  object ForStatement {
    def unapply(node: dom.ForStatement) = {
      val initializers: List[dom.Expression] = node.initializers
      val expression: dom.Expression = node.getExpression
      val updaters: List[dom.Expression] = node.updaters
      val body: dom.Statement = node.getBody

      Some(initializers, Opt(expression), updaters, body)
    }
  }

  // Scala niftiness #5: pattern matching
  def matchDemo(node: dom.ForStatement) = node match {
    case ForStatement(inits, _, _, _) if inits.size > 1 =>
      println("Multiple inits")
    case ForStatement(Nil, Some(InfixExpression(lhs, Op("=="), rhs, Nil)), Nil, body) =>
      println("Why not a while loop?")
    case ForStatement(_, None, _, _) =>
      println("No expression")
    case _ =>
      println("If all else fails")
  }
}
```

Scala Pain Points

- Multiple constructors
- Varying variance semantics
- Results of assignment
- Namespaces
- Absence of unexceptional gotos (no break, continue, or labeled statements)
- Intersection of inheritance and statics
- And many more!

Java	Scala
Arbitrarily many independent constructors	One primary - auxiliaries must call this(...) as first instruction
Any constructor can call any available superconstructor	Only the primary can call super
Early return is legal	No returning from constructors

```
1.  // this class cannot be directly expressed in Scala
2.  public class Sub extends Superclass {
3.    public Sub(int x)          { super(x); }
4.    public Sub(String s)       { super(s); }
5.    public Sub(String[] xs)    { super(xs); }
6.    public Sub(int x, int y) { this(x + y); }
7.
8.    public int method1() { return 5; }
9.    public static int method2() { return 10; }
10. }
```

```scala
// Scala niftiness #6: traits
// Everything but the constructor into a trait...
abstract trait Sub extends Superclass {
  def method1 = 5
}

// ...and one class per primary constructor mixes it in
class Sub0(x : Int) extends Superclass(x) with Sub {
  def this(x: Int, y: Int) = this(x + y)
}
class Sub1(s : String) extends Superclass(s) with Sub
class Sub2(xs : Array[String]) extends Superclass(xs) with Sub

// Scala niftiness #7: objects
// objects are language-supported singletons - an object with
// the same name as a class is its "companion object", and the
// class and object can access one another's private data
object Sub {
  def apply(x : Int) = new Sub0(x)
  def apply(x : Int, y: Int) = new Sub0(x, y)
  def apply(s : String) = new Sub1(s)
  def apply(xs : Array[String]) = new Sub2(xs)

  // note static method in companion object
  def method2 = 10
}

// Scala niftiness #8: function objects
// any apply method can be accessed like a function
// in the end we saved four characters over calling "new"
val s1 = Sub(5)
val s2 = Sub("hello")
val s3 = Sub(Array("one", "two"))
```

Namespaces

Unfortunately the following is legal Java:

```
 1.  package x;
 2.
 3.  class x     { public int x = 2; }
 4.  class Base { public int x = 3; }
 5.  public class Name extends Base {
 6.    public int x = 5;
 7.    public int x() { return 7; }
 8.    public Name(int x) {
 9.      x = 11;
10.      System.out.println(super.x * this.x * x * x() * (new x()).x);
11.    }
12.  }
13.
```

In Scala, there are only two namespaces (compared to Java's four), mutable variables cannot be overridden in subclasses, and method parameters are immutable.

Suffice to say that preserving maintainability will pose a challenge.

Arrays and Lists

- In Java, arrays are covariant and lists are invariant
- In Scala, lists are covariant and arrays are invariant
- Gratuitous? Perverse? Malicious? No, it turns out:

- Covariant arrays are not typesafe, thus Java's runtime checks and ArrayStoreExceptions. Scala says: no.
- Scala lists can safely be covariantly typed because they are immutable.

Java code expects arrays to be covariant. What to do?

```
1.  // Scala niftiness #9: interactive interpreter
2.  scala> val arrayOfStrings: Array[String] = Array("a", "b", "c")
3.  arrayOfStrings: Array[String] = Array(a, b, c)
4.
5.  // Since arrays aren't covariant, this fails
6.  scala> val arrayOfObject: Array[AnyRef] = arrayOfStrings
7.  <console>:5: error: type mismatch;
8.   found   : Array[String]
9.   required: Array[AnyRef]
10.        val arrayOfObject: Array[AnyRef] = arrayOfStrings
11.                                           ^
12. // Implicits: feel the power
13. scala> implicit def covArray[T, U >: T](xs: Array[T]): Array[U] =
14.        xs.asInstanceOf[Array[U]]
15. covariantArrays: [T,U >: T](Array[T])Array[U]
16.
17. // Can you hear me now?
18. scala> val arrayOfObject: Array[AnyRef] = arrayOfStrings
19. arrayOfObject: Array[AnyRef] = Array(a, b, c)
20.
21. // Doesn't happen the other way because U >: T does not hold
22. scala> val newArrayOfStrings: Array[String] = arrayOfObject
23. <console>:7: error: type mismatch;
24.  found   : Array[AnyRef]
25.  required: Array[String]
26.        val newArrayOfStrings: Array[String] = arrayOfObject
27.
```

If we depend on implicits here, we incur edge cases and ambiguities - for instance:

```
1.  // note: java.lang.Object and scala.AnyRef are equivalent
2.  class Overloaded {
3.    def aMethod(objs: Array[AnyRef]) = { ... }
4.    def aMethod(obj: AnyRef) = { ... }
5.  }
6.
7.  // In java, if we call aMethod with any array it
8.  // will always call the first variant, because any
9.  // Array is also an Array[Object]
10. //
11. // In scala, the first variant will only be called for an
12. // Array[AnyRef].  Our implicit conversion won't help
13. // us because implicits only activate after type checking
14. // fails, and since Array descends from AnyRef, that
15. // stage will not be reached.
16.
```

Implicits are a convenient hammer for making sweeping semantic changes, but in the end they'll be eliminated wherever possible.

Scala on the JVM

- http://openjdk.java.net/projects/mlvm/subprojects.html
- ..most of these are potentially useful in Scala - faster everything, **tail call optimization**, interface injection, autonomous methods, and a pony.
- Many of us want the same things, especially the functional language people.

Side note on Reifiable Types

- I would suppose most JVM languages have one or more ambitions hampered by type erasure
- Scala initially adopted Java's erasure model, but perhaps that's not settled?
- Recent Scala versions have a (still undocumented I think) feature: "A Manifest[T] is an opaque descriptor for type T."
- So since the vibe I get is that it's not coming to the JVM anytime soon, at least we can do this:

```scala
import scala.reflect.Manifest

object Demo {
  // as you are all aware, erasure makes this impossible
  def idfail[T](xs: List[T]) = xs match {
    case _: List[String] => "list of smurfs"
    case _ => "list of gadzooks!"     // unreachable
  }

  // this requires a recent version of scala
  def id[T](xs: List[T])(implicit m: Manifest[T]): String =
    if (m <:< Manifest.classType(classOf[String])) "list of strings"
    else if (m <:< Manifest.classType(classOf[Int])) "list of ints"
    else if (m <:< Manifest.classType(classOf[List[_]]))  "list of lists"
    else "mystery list: " + m.erasure.toString

  def go = {
    val ints = List(1, 2, 3)
    val strs = List("a", "b", "c")
    println("idfail: " + idfail(ints))
    println("idfail: " + idfail(strs))
    println("id: " + id(ints))
    println("id: " + id(strs))
    println("id: " + id(List(ints, strs)))
  }
}

scala> Demo.go
idfail: list of smurfs
idfail: list of smurfs
id: list of ints
id: list of strings
id: list of lists
```

Classfile Format

- The shiniest features of Scala are implemented with lots and lots of highly redundant little classes
- Scala's own performance challenges are more than sufficient, thanks
- This example not contrived - a Java project rewritten in Scala could easily see a 10x or greater increase in classfiles

```java
public enum Planet {
    MERCURY (3.303e+23, 2.4397e6),
    VENUS   (4.869e+24, 6.0518e6),
    EARTH   (5.976e+24, 6.37814e6),
    MARS    (6.421e+23, 3.3972e6),
    JUPITER (1.9e+27,   7.1492e7),
    SATURN  (5.688e+26, 6.0268e7),
    URANUS  (8.686e+25, 2.5559e7),
    NEPTUNE (1.024e+26, 2.4746e7),
    PLUTO   (1.27e+22,  1.137e6);

    private final double mass;   // in kilograms
    private final double radius; // in meters
    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
    }
    public double mass()   { return mass; }
    public double radius() { return radius; }

    // universal gravitational constant  (m3 kg-1 s-2)
    public static final double G = 6.67300E-11;

    public double surfaceGravity() {
        return G * mass / (radius * radius);
    }
    public double surfaceWeight(double otherMass) {
        return otherMass * surfaceGravity();
    }
}
```

Left: A classic Java enumerated type
Below: Plausibly idiomatic Scala translation
lifted from an authentic blog

```scala
case object MERCURY extends Planet(3.303e+23, 2.4397e6)
case object VENUS extends Planet(4.869e+24, 6.0518e6)
case object EARTH extends Planet(5.976e+24, 6.37814e6)
case object MARS extends Planet(6.421e+23, 3.3972e6)
case object JUPITER extends Planet(1.9e+27,   7.1492e7)
case object SATURN extends Planet(5.688e+26, 6.0268e7)
case object URANUS extends Planet(8.686e+25, 2.5559e7)
case object NEPTUNE extends Planet(1.024e+26, 2.4746e7)
case object PLUTO extends Planet(1.27e+22,  1.137e6)

// mass in kilograms, radius in meters
sealed case class Planet( mass: double, radius: double ){
    // universal gravitational constant  (m3 kg-1 s-2)
    val G = 6.67300E-11
    def surfaceGravity = G * mass / (radius * radius)
    def surfaceWeight(otherMass: double) = otherMass * surfaceGravity
}
```

And finally: the sadness

```
leaf:enum paulp$ ls java ; du -h java
Planet.class
4.0K    java

leaf:enum paulp$ ls scala ; du -h scala
EARTH$.class    MARS$.class     NEPTUNE$.class  Planet$.class   URANUS$.class
EARTH.class     MARS.class      NEPTUNE.class   Planet.class    URANUS.class
JUPITER$.class  MERCURY$.class  PLUTO$.class    SATURN$.class   VENUS$.class
JUPITER.class   MERCURY.class   PLUTO.class     SATURN.class    VENUS.class
80K     scala
```

Observation re: Functional Programming

- Imminent ascension of FP predicted...
- ...as it has been every few years since I started programming.
- It might be true this time, honest!
- FP mainstays like full tail call optimization and first class continuations in the JVM would mean more FP mindshare. Or at least happier Scala (and Clojure) programmers.
- FIN