Implementation of JVM-based languages support in IntelliJ IDEA

Ilya Sergey
ilya.sergey@jetbrains.com

Abstract

This paper describes several examples of usage of two languages, compilable to Java bytecode, namely, Scala and Groovy. We consider some functional and dynamic language constructs they bring to standard process of Java application development. For that purpose we appeal to the language analysis and IDE development process as comprehensive examples demonstrating the benefits of Scala and Groovy usage.

1 Introduction

Java programming language holds a palm of one of the most popular technologies for developing various types of applications for different industries and areas. Nevertheless in some situations it is insufficient of standard Java constructions which forces us to produce a fair amount of boilerplate code. In this work we will show how to use possibilities of interoperability between different languages in a full production taking two JVM-based languages, Scala and Groovy, as an example.

When implementing languages support in IDE, we deal with syntactic and semantic analysis of tree-like structures. For these goals Scala provides such useful language constructions as higher order, nested, parametric and structural polymorphic functions which could be used to make parser code both shorter and conciser. Haskell-style list comprehensions simplify creation of filters for semantic elements of different kinds. Implicit conversions make the wrapper creation more transparent and pattern matching helps pick elements with similar structure.

Groovy loses in performance-critical tasks but it perfectly suits for testing purposes as it can expand classes functionality at runtime, that eliminates tedious Java reflection API usage. Another use case is simple generation of build scripts and XML documents via elegant implementation of builder pattern.

In general, it would make sense to allow use of Scala and Groovy to make existing Java applications code more elegant and maintainable. In opposite direction Scala programs could take benefit using well-known Java legacy such as, for example, numerous UI libraries.

2 Scala in IDE development

2.1 IDE development stages

Process of Integrated Development Environment creation bears a resemblance to compiler development. It has the same stages such as lexical analysis, syntactic analysis and at later time various semantic verifications, for instance, type checking, to resolve as many semantic conflicts as possible. But from other side of view, IDE as a program is much more exacting to performance of such analysis's. They must be more or less incremental, and they must not be broken after first user's error as many auto-generated compilers.

That why *IntelliJ IDEA*'s language API contains rich abstractions to make process of development easier. Source files are represented by special tree-like entity named Program Structure Interface (PSI). It is nothing but hierarchy of wrappers around program tree nodes, received from

parser. But PSI encapsulates all logic related to analysis and modification of program structure, it serves as intermediate level between text representation of a program document and IDE logic, such as refactorings, for example. Despite all opportunities for plugin development this API demands big amount of code, which could be decreased if we have appropriate functional constructions.

We consider some of *IntelliJ IDEA* language API elements and try implement them in functional style. For this we'll take Scala Programming language [1] as a magnificent tool to demonstrate some patterns.

2.2 Lexer's look-ahead

One of the most important abstractions to create parser in IDEA is so-called `PsiBuilder`, which is charged with lexer advances and creation of program tree nodes. Suppose, we deal with with some $LL(k)$-grammar where k is known. Then we would like to have method called `lookAhead(PsiBuilder builder, int k)` which could give us next k tokens from input stream. In Java this code could look as follows:

```
static TokenType[] lookAhead1(PsiBuilder builder, int k) {
  Marker marker = builder.mark();
  TokenType[] tokens = new TokenType[k];
  for (int i = 0; i < k; i++) {
    tokens[i] = builder.eof() ? null : builder.getTokenType();
    builder.advanceLexer();
  }
  marker.rollbackTo();
  return tokens;
}
```

Looks not so bad, but imagine we have to analyse these k tokens further. So we're facing the prospect of infinite checks of every token in our array. Just the time to recall Scala's high-order functions and pattern matching. Same code written on Scala looks much better.

```
def lookAhead2(n: Int)(builder: PsiBuilder) = {
  val marker = builder.mark
  val tokens = for (i <- (1 to n) if !builder.eof
                    token <- builder.getTokenType) yield {
    builder.advanceLexer
    token
  }
  builder.rollbackTo
  tokens
}
```

Infix expressions in Scala are nothing but method invocations. The `(1 to n)` expression invokes implicit function to construct `Iterable` instance by two integers. `lookAhead2` function returns instance of `scala.Seq` which represents sequence class and can be used in pattern matching[1]:

```
lookAhead2(3)(builder) match {
  case Seq(VAR, ID, ASSIGN, _*) => Assignment(builder)
  case Seq(NUMBER, _*) => NumLiteral(builder)
  case Seq(CLASS, ID, _*) => ClassDefinition(builder)
  case _ => throw new ParseError("Unknown token sequence")
}
```

[1]Notice, that type of `lookAhead2` function is inferred from its right part. Type of returned value must be specified explicitly only when there are recursive calls or `return`-statements in the body of function.

In the last case we check any other variant using wildcard pattern which matches any value. Sequence wildcard pattern `_*` matches any number of elements within a sequence, including zero elements. We divided parameter set in signature of `lookAhead2` function into two clauses. It has been done to provide possibility to *curry* our function by its last parameter.

Suppose all non-terminals in our grammar are represented by *singleton object* of special kind and for some non-terminal we want to fix our k by some N. Then we may use mix-in functionality and create more specific `lookAheadN` by making our `lookAhead2()` function partially applied.

```
trait ParserNode3 {
  def lookAhead3 = ParserUtils.lookAhead2(3) _
}

object CodeBlock extends ParseNode3 {
  def apply(builder: PsiBuilder) = lookAhead3(builder) match {
    // some cases
  }
}
```

The final step will let us eliminate builder as an argument for `lookAhead` function. For this purpose we will change `CodeBlock` from object to *case class*.

```
case class CodeBlock(builder: PsiBuilder) extends ParseNode3 {
   implicit def unit2Builder(u: Unit): PsiBuilder = builder
   def parse = lookAhead3() match {
     // some cases
   }
}
```

We marked our parser node class with `case` modifier which let us avoid `new` keyword before new instance declaration, but all parser's work will be done now by method `parse`. Here we defined auxiliary implicit function `unit2Builder` which converts argument of `Unit` type to `PsiBuilder`, passed as constructor argument. When we call `lookAhead3` with parentheses, we pass an instance of `scala.Unit` type to it as an argument, which is converted further implicitly by `unit2Builder` to `PsiBuilder` instance. Of course, we could define new method, for example `def lookAhead4 = lookAhead3(builder)` without any parameters. But by agreement it would be better if all methods which have side effects like our `lookAhead4` will have non-empty parameter clause.

2.3 Implicit conversions and Pattern Matching

We have already looked at user-defined conversions, implemented by Scala implicit functions. They found one more interesting application in our project, concerned to Scala support in *IntelliJ IDEA*. We had to distinguish, which of compiled `.class`-files were got from Scala traits, classes or objects to get their variables and methods and work with them. Picking any `.class`-file from the set of compiled files, we can deduce what kind of Scala entity it was compiled from using some heuristics, related to auxiliary functions. In such code it would be very pleasant not to process separately case, when some entity (trait, object etc.) was received from `.class`-file or from source file. That's why appropriate classes appeared.

```
case class TypeDefinition(methods: Seq[Method], fields: Seq[Field])

case class ScObject(methods: Seq[Method], fields: Seq[Field])
     extends TypeDefinition(methods, fields)
case class ScClass(methods: Seq[Method], fields: Seq[Field])
     extends TypeDefinition(methods, fields)
case class ScTrait(methods: Seq[Method], fields: Seq[Field])
     extends TypeDefinition(methods, fields)
```

```
implicit def cls2TypeDef(cls: ClsClass): TypeDefinition = {
  //Logic to obtain appropriate representation
}
implicit def src2TypeDef(cls: SourceClass): TypeDefinition = {...}
```

Now if we need to work with „pure“ TypeDefinition, not thinking about its real entity, we may use simple pattern matching. Implicit conversion will do all dirty work for us. For example, if we want process only class fields or object methods we can write:

```
someEntity: TypeDefinition match {
  case o@ScObject(m, _) => processObjectMethods(o, m)
  case c@ScClass(_, f) => processClassFields(c, f)
  case _ =>
}
```

We must specify type of `someEntity` explicitly before matching to invoke our implicit conversions. `@` symbol is used as in Haskell to bind our pattern with named value. And, of course, wildcard patterns in case classes are also available.

2.4 Some words about higher order functions

Higher order function which are typical for functional programming proved to be very handy to work with some syntactical construction in our Scala plugin for *IntelliJ IDEA*. We have already made an acquaintance with pattern matching and appropriate name binding. Suppose we want to get all subpatterns of any pattern that bind any identifiers. For example the pattern below binds five identifiers.

```
case t@(Node(n), s: String, List(_, l, ls @ _*)) => ...
```

So, we bound `t` as tuple, `n` as parameter of case class `Node`[2], `s` of type `String`, `l` as second element of some `List` and `ls` as sequence representing the tail of the same list. In our IDE background we represent syntactic construct of such type by appropriate class.

```
class ScPattern extends Node {
  def subpatterns: Seq[ScPattern] = ...
  def bindedName: String = ...
}
```

Now we can obtain **all** subpatterns of root pattern only by three lines of code:

```
def subpatterns: Seq[ScPattern] = this ::
  childrenOfType(classOf[ScPattern]).foldLeft(Nil,
    (x, y) => x ++ y.subpatterns)
```

To get all bindings we have to filter this sequence.

```
def bindings = subpatterns.filter(_.bindedName != null)
```

In a word, we replaced standard recursive tree bypass by high-orderer function `foldLeft` of trait `scala.Seq`. To filter only those patterns which bind some identifier we used method closure with placeholder syntax, using _ symbol instead of single argument. Scala type inference system will determine its type as type argument of appropriate instance `scala.Seq`, namely `ScPattern`. That's why invocation of `bindedName` of placeholder parameter is legal.

[2] or appropriate extractor

2.5 Sequence comprehensions and filters

One of the most useful features of functional languages like Haskell is *list comprehensions* which could be used to filter some sequences and get new ones. Classical example of list comprehension is the shortest quicksort implementation[3].

```
def qsort[T <% Ordered[T]](l: List[T]): List[T] = l match {
  case Nil => Nil
  case x :: xs => qsort(for (e <- xs if e < x) yield e) :::
    List(x) ::: qsort(for (e <- xs if e >= x) yield e)
}
```

We have to mention some important details in example above. Firstly, `Nil` is an object that extends `List[Nothing]` and taking into consideration covariance inheritance annotation and the fact that `Nothing` is placed at the bottom of Scala's type hierarchy, we may return it as instance of `List`, parametrized by any type. Secondly, let's look at that strange sign `<%`. It denotes visible bound for implicit conversions that makes from given type `T` its „ordered wrapper". `<%` is like type bound but it's indicating, that `qsort()` function may be applied only for arguments of such type `T` which can be implicitly converted to type `Ordered[T]`. It applies to those types that are subtypest of trait `Ordered` and has necessary methods `<()` and `>=()` to be compared with each other.

There are many situations where sequence comprehensions may be elegantly replaced by filters or loops. But not in the case when we need to treat elements of two or more sequence.

Let's discuss such useful IDE feature as keyword completion. To make it more smart we must analyse semantical context and choose only those keywords that could appear in this context in correct code. It's obvious, that in Java we cannot suggest keyword `interface` after keyword `double` has been typed by user. That's why we build a set of probably contexts by editor's caret position. Every keyword is represented by semantic „variant", containg all information about its significance. In common case we have to check all variants for application for every one of built contexts. In those cases when some variant is applicable to some context we have to add appropriate keyword to the set of keywords to be suggested as variants for completion. On Scala we may describe this tangled operation by `for`-comprehension.

```
val keywordsToSuggest = for (c <- buildContexts
                             v <- KeywordVariants
  if v.isApplicable(c)) yield v.keyword
```

2.6 None object instead of null

That's very common situation when some method returns `null` object instead of expected value. That's why we have to write endless sequences of checks for `null`. *IntelliJ IDEA* API provides two special kinds of Java method annotations, namely `@NotNull` which allows not to check returning value for `null`, and `@Nullable` annotation to make developer check returned value. Our modified Java compiler checks written code to conform to these annotations. For example non-checked result of `@Nullable`-marked method invocation produces a warning message during compilation.

Developing our Scala plugin we found that it would be much more reasonable to make methods returning values of some type `T` (and probably returning `null` value) return value of type `scala.Option[T]`. Haskell adepts must be familiar with this type, it's nothing else but Haskell's `Maybe`. In Scala type `scala.Option[T]` has, as was expected, two subtypes.

```
case final class Some[+T] (x: T) extends Option[T]
case object None extends Option[Nothing]
```

That's why we rejected unsafe method invocations followed by checking for `null` value.

[3]Here we deal with lists, that's why we can use `:::` operator to concatenate resulting sequences.

```
String dangerous() { ... }
if (dangerous() != null) {
  String d = dangerous();
  ...
} else {
  ...
}
```

We replaced them by safe pattern matching on instances of `scala.Option[T]` type.

```
def safe(): Option[String] = ...
match safe(){
  case Some(x) => ... // use x
  case None => ...
}
```

3 Testing with Groovy

Groovy is another object-oriented language compilable to Java byte-code. In contrast to Java or Scala it is dynamic and has features similar to those in other popular dynamic languages, such as Ruby, Python or Perl. Of course programs written on Groovy are fully compatible with Java libraries. Furthermore using mechanism of Java stub generation you may write cross-referenced program on both languages - Java and Groovy and you don't need to think about order of compilation. Groovy-to-Java stubs had been implemented for the first time in JetGroovy plugin for *IntelliJ IDEA* and after they were added as a feature to Groovy compiler itself.

As it was said before, Groovy is not the most pertinent language for IDE development because of its performance problems. They are related to big amount of auxiliary method invocations to provide dynamic features at the base of JVM. In spite of this fact many classical patterns of object-oriented programming look much more simpler and expressive being written on Groovy. That's why we used Groovy to test our applications.

3.1 Test project creation and Builder pattern

Builder pattern is well-known conception to separate construction of a complex object from its representation. There is important abstraction In *IntelliJ IDEA* testing API named `IdeaTestFixure`. It encapsulates program logic related to Project creation, adding to it references to Java SDK, user-defined libraries and other settings. It's necessary to reproduce the whole environment's behaviour and test some functionality like, for instance, name binding.

Groovy provides very elegant syntax to implement builder pattern. It's based on some principal moments. At first, you can pass arguments by name to some methods of Groovy classes. In fact, this is `java.util.Map` as their first parameter. Then all arguments, passed to this method by name, will be collected in this map with those names as keys. After that they can be got by these names inside method body. Secondly, no one restricts these arguments to be of any type. For example, they may have type `groovy.lang.Closure` and involve some builder logic. Now, to create test project we have to implement class `ProjectBuilder`.

```
class ProjectBuilder{
  def myProject
  def buildProject(Map args, Closure cl) {
      if (args.name != null) myProject.name = args.name
      // Other named arguments processing
      cl.setDelegate this
      cl.call()
      myProject
```

```
  }
  def addModule(Map args, Closure cl) {...}
  def addJdk(Map args) {...}
  // other builder methods
}
```

It's very important that in given parameter `cl` of type `groovy.lang.Closure` of `buildProject()` method we replaced `cl`'s delegate by `ProjectBuilder` class itself. Now we can invoke methods of this class from closure created *outside* of it. So, to create test project we have to invoke `buildProject()` method in the following way.

```
def builder = new ProjectBuilder()
def testProject = builder.buildProject(name: "TestProject",
                                       type: "JavaProject") {
  addModule(name: "TestModule") {
    addLibrary(path: "/home/user/my_lib_path")
  }
  addJdk(path: "/home/user/jdk_home")
}
```

In example above we pass two named arguments, `name` and `type`, and one instance of `groovy.lang.Closure` (code block in curly braces) into `buildProject()` method. Of course other methods like `addModule()` can also take closures as arguments. That's why the whole creation procedure is nothing but several nested closure invocations.

3.2 Null object and safe dereferencing

Let's return again to the methods returning `null` value. In test we have no so intense need to check every possible „dangerous“ reference for `null`. For that cases Groovy provides special safe dereferencing operator „`?.`“. Usually it's assumed, that result of using the `null` value should semantically be equivalent to doing nothing. Reference expression with safe dereferencing operator will not throw `java.lang.NullPointerException` in case when referenced expression points to `null`, but it will return `null`. The code below will print line „null“.

```
String nullString = null
println nullString?.length()
```

Another way to avoid numerous null-checks before referencing fields or invoking methods of some object is to use itself in the condition of conditional operator. This feature named „Groovy truth“ consists in implicit casting of some object values to boolean value `true` or `false`. For example non-empty collection or string instance will be processed as `true`. The most part of objects references will be casted to `true` value if they are non-null. Hence we can replace allchecks by code like below.

```
if (nullString) {
  println nullString.length()
}
```

3.3 Regular expressions and properties setting

Let's imagine we have written a test for numerous settings of code formatter. It would be more reasonable to place all of them into some file and read immediately before test. But after reading them we must set up appropriate static fields of `CodeStyleSettings` class. We could do it but numerous if-else constructions or by `switch` operator if it would support switching by strings in Java. But it does not. So, let's use power of dynamic language Groovy. At first, place our properties to be set up to some file.

```
<option>BRACE_STYLE=END_OF_LINE</option>
<option>FINALLY_ON_NEW_LINE=false</option>
[testdata]
```

After that in our test class declare regular expression pattern to match our settings:

```
static OPTION_START = '<option>'
static OPTION_END = '</option>'
static PATTERN = /$OPTION_START\w+=(true|false|\d|\w+)$OPTION_END\n/
```

So, it's time to get our properties from text.

```
testText.eachMatch(PATTERN) {match ->
  List tokens = match[0].trim().
                replace(OPTION_START, "").replace(OPTION_END, "").tokenize("=")
  [property, value] = [tokens.get(0), tokens.get(1)]
  setSettingsProperty(property, value)
}
```

According to Groovy 1.6 specification we can use assignments of list expressions to assign multiple values simultaneously. It remained only to set appropriate fields of `CodeStyleSettings` class.

```
def setSettingsProperty(property, value) {
  if (value ==~ /(true|false)/) {
    mySettings."$property" = Boolean.parseBoolean(value)
  } else if (value ==~ /\d/) {
    mySettings."$property" = Integer.parseInt(value)
  } else {
    mySettings."$property" = mySettings."${value.toString()}"
  }
}
```

Here we use so-called GString with dollar sign inside to inject defined variables directly into string expression.

4 Conclusion

We presented an implementation of solution for some usual problems in IDE development using Scala and Groovy languages to take advantage of their functional or dynamic features correspondingly. The resulting code looks more elegant and readable. We used these languages in the process of development language plugins for *IntelliJ IDEA* which was positioned at the beginning as IDE only for Java language. But now with its plugins for Scala and Groovy languages it becomes a powerful tool to develop properly big projects on all of these languages concurrently.

References

[1] M. Odersky, *The Scala Language Specification, Version 2.7*, May 5, 2008

[2] B. Emir, M. Odersky, J. Williams, *Matching Objects With Patterns*, January 2007.

[3] *Developing Custom Language Plugins for IntelliJ IDEA*, available at http://www.jetbrains.com/idea/plugins/developing_custom_language_plugins.html

[4] *Nullable How-To*, available at http://www.jetbrains.com/idea/documentation/howto.html

[5] D. Koenig, A. Glover, P. King, G. Laforge, J. Skeet, *Groovy in Action*, January, 2007, 696 pages

[6] N. Ford, *„Design patterns“ in dynamic languages*