Automatic refactorings for Scala programs

Taming multi-paradigm code

Ilya Sergey Dave Clarke
DistriNet, Katholieke Universiteit Leuven
{ilya.sergey, dave.clarke}@cs.kuleuven.be

Alexander Podkhalyuzin
JetBrains Inc.
alexander.podkhalyuzin@jetbrains.com

Abstract

Scala is a programming language that combines the object-oriented and functional programming paradigms. Dependent types, higher-order functions and implicit conversions bring new ideas and challenges when implementing code refactoring. In this paper, we give an overview of the automatic refactorings for Scala programs offered by the IntelliJ IDEA programming environment. We consider the main differences between well-known automatic refactorings implemented for Scala with their Java analogues, and give short descriptions of the pitfalls encountered and underlying techniques. Finally, we provide a short survey of refactorings that have not yet been implemented, but might be useful for practical software development in Scala.

Keywords Scala, refactoring, IDE

1. Introduction

In software engineering, refactoring source code means improving it without changing its overall behavior; it is sometimes informally referred to as cleaning up the code. Refactoring neither fixes bugs nor adds new functionality, although it might precede either activity. Rather, it improves the understandability of the code, changes its internal structure and design, and removes dead code. The canonical reference on code refactoring is Martin Fowler's book [1]. A lot of classifications of refactorings exist. Depending on their effect, techniques are distinguished as being for better code abstraction, modularization, or naming and location improvement. Advances in integrated development environments (IDE) enable "automated refactoring", i.e. automatic code transformations performed by an intelligent tool in accordance with the programmer's intention. As the object-oriented paradigm is the most popular in modern industrial programming, it is hard to imagine an IDE for an object-oriented language without automated refactorings such as automatic renaming of methods, fields and variables, extracting/inlining methods etc. The domain of automatic refactoring in object-oriented frameworks was pioneered by Opdyke [9] in his thesis work, where he gave the main ideas on implementing refactorings and on the basic notion of the correctness of a refactoring as a behaviour-preserving program transformation.

In this paper we discuss different code refactorings for programs written in Scala. Scala is a fusion of object-oriented and functional programming paradigms. Thus classical code transformations for Java programs must be reconsidered and reformulated in appropriate terms before they can be implemented as automatic refactorings for Scala. Even seemingly well-known ones such as *introduce variable* or *extract method* become much more challenging to implement in the presence of closures and implicit conversions.

The remainder of this paper is structured as follows. Sections 2, 3 and 4 describe some automatic refactorings that have already implemented in the Scala plugin for the IntelliJ IDEA programming environment [3]. Section 2 describes issues related to automatically maintaining import statements in Scala files. In Section 3 we discuss the introduction and inlining of local variables. Section 4 gives details about the implementation of the *extract method* refactoring. Section 5 offers a short list of refactorings that have not yet been implemented in IntelliJ IDEA, but might be nonetheless interesting for Scala practitioners. Section 6 discusses related work on refactorings for functional programs and Section 7 concludes.

2. Automatic class import optimizer

Even simple functionality such as the automatic addition or removal of import statement into a file may lead to different design solutions.

2.1 Adding missing imports

Goal: Add missing import for an unresolved type identifier
Challenge: Import statements are context-sensitive

The simplest logic for automatic import statement addition in Java works as follows. First choose the right qualified class name from the set of possible variants. The IDE inserts it to the list of import statements at the beginning of the file, sorting the list just after the new import statement has been added. Unfortunately, this approach does not work for Scala, since it has *flow dependencies* between imports. We illustrate this with the following example. The code fragment in the Figure 1 gives an example of an unresolved reference name `BitSet`. Different actual implementation of the `BitSet` class may be imported. Assume we decide to use the default one from the `scala.collection` package. There are at least two possible places to add the statement **`import`** `scala.collections.BitSet`. The site marked with the *place 1* comment results in the expected semantics, but if the import statement is put at *place 2*, the name `BitSet` will refer not to the default implementation but to the one from the `TestCollection` object.

One possible solution to this problem is to always use the imported name with the most specific qualifier prefix. On the other hand this approach does not work well if one needs to import an identifier from a field or local final variable. The currently implemented algorithm for adding new imports works as follows:

```
// place 1
object TestCollections {
  object scala {
    object collection {
      trait BitSet[K]
    }
  }
}

import TestCollections._
// place 2

val set =  new BitSet[Int]
```

Figure 1. Different places to put a missing import statement

```
class A
class B
object MyConversions {
  implicit def a2b(a: A): B = new B
}

def testB(b: B): Any = {/* some code */}

import MyConversions._

val a = testB( new A )
```

Figure 2. Meaningful import of an implicit conversion function

1. Starting from the unresolved identifier, go bottom-up, collecting all possible providers of type definitions: immutable variables, objects or packages;
2. Chose a scope to add the import statement into: the closest code block or the most general code block, according to user settings; and
3. Inspect the context of the chosen place to resolve possible naming conflicts (as in the Figure 1) and adjust the prefix of names if necessary.

2.2 Optimize imports

Goal: Remove unused import statements
Challenge: Implicit conversions demand additional checks

Another popular refactoring is to removing unused import statements. In Java the corresponding algorithm is straightforward:

1. Resolve all references in the file using a bottom-up traversal;
2. For every resolved reference keep the set of import statements used during its resolution; and
3. Remove imports that do not appear in the set of *previously registered* ones.

In Scala the procedure of unused imports removal is more complicated due to the presence of implicit conversions. In the example in the Figure 2 the import statement `import MyConversions._` does not participate directly in the reference resolution process, so it would be removed according to the above algorithm. However, after its removal the code becomes incorrect, since the removed statement was "responsible" for the implicit conversion of the grayed expression `new A` into the instance of type `B`, but the code to perform this conversion would no longer be available.

The Scala version of the algorithm not only checks that there is a definition for every identifier in a file, but also performs type checking for all expressions and tracks imports used for *implicit conversions and parameters*. For instance, the import statement `import MyConversions._` in Figure 2 will be marked as *used* as it brings the necessary implicit function `a2b` into the context. These import statements are marked as *used* and will not be removed by the optimizer.

3. Introduce and inline variable refactorings

The *introduce variable* refactoring gives rise to a new coding style: *write-introduce*. Normally, Java programmers working in an IDE do not declare a variable with a common tedious preamble such as `final Map<String, Object> map = new ...`. Instead, they write something like `new HashMap<String, Object>` and press a predefined keystroke to create a new variable with this expression as an initializer. Finding a new name for a variable is also not a big problem: it is automatically suggested by the IDE based on the type of the expression and the current bound names in context to prevent any conflicts.

The dual operation of *introduce variable* is the so-called *inline variable* refactoring.

3.1 Introduce variable refactoring

Goal: Extract an expression to a new variable or parameter
Challenge: It may have different types depending of the context

The Figure 3 shows an example of the *introduce variable* dialog in IntelliJ IDEA. Many options are available for the selected expression. For example, it is possible to switch between mutable and immutable variable creation: the `val` or `var` modifier will be added to the beginning of the declaration, respectively. The *replace all occurrences* checkbox is available if identical expressions, modulo white spaces and comments, are found in the context.[1]

The most interesting part of this dialog is the checkbox *specify type explicitly* and its corresponding drop-down list. Since the Scala compiler is powerful enough to infer the type of any expression lacking explicit returns and recursive function calls, there is almost no need to use this option for local variables. But it is essential for function parameters which must be annotated with explicit types.

It is logical to assume that the type which the newly introduced parameter should be annotated with the most specific type of the selected expression. But sometimes it is more reasonable to introduce a parameter with more *general* type. This is why the IDE uses the *variable of type* drop-down to suggest the most precise type as well as any of its supertypes. This is not all, as the Scala language specification [8, Section 6.25] describes multiple implicit conversions, as well as user-defined implicit conversions, and those imported from the `scala.Predef` object, all of which need to be considered.

On the other hand, it is also not such a good idea to suggest all the possible supertypes and codomains of implicit conversions as possible parameter types. When selecting types to suggest, the IDE must take into account the context in which the selected expression is used. Figure 4 gives an example of some code with an expression of type `String` to be introduced. Since it is used as the parameter of type `java.lang.Serializable`, the set of suggested types must be bound by it. Thus the final set of variants should consist of types `String`, `Serializable` and the codomains of implicit conversions which conform to them.

The mentioned context of the expression is defined by its position: passing an expression as a parameter or using it as a method call receiver brings additional restrictions on the possible types of

[1] The duplicate detection algorithm is being improved to find identical code fragments modulo bound closure parameters

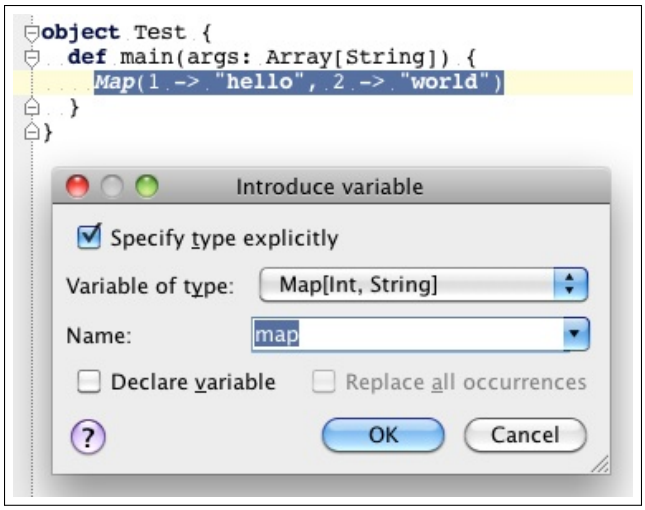

Figure 3. IntelliJ IDEA's *introduce variable* refactoring dialog

```
def operate(s: Serializable) = {...}

def test = {
  operate( "Hello, " + "World!" )
}
```

Figure 4. Introducing a string expression as a parameter

a parameter. In other words, all the supertypes and implicit conversions' codomains must conform to the context's type bounds. If we replace all the duplicates by the introduced parameter, all the restrictions must be collected from *all contexts* and applied to the set of variants.

3.2 Inline variable refactoring

Goal: Inline all occurrences of a variable
Challenge: Computing reaching definitions of local variables

As most local variables in Scala are immutable, it is relatively easy to inline them. The IDE just replaces all the occurrences of the immutable variable with its value. Nevertheless, *mutable* variables cause some problems as they can be captured by closures. In fact, the problem of inlining local variables is also reduced to the problem of computing reaching definitions. Figure 5 gives an example of code for which it is hard to reason about reaching definitions of the variable `a`, especially if there is nothing concrete known about the behaviour of the function `foo`. Thus in this example there is more than one possible reaching definition for the grayed occurrence of the variable `a` and the IDE conservatively refuses to inline it with some value.

```
var a = 1
val cl = {(); {(x: Int) => a = x}}
foo(cl)
println( a )
```

Figure 5. Local mutable variable is modified within the closure

4. Extract method refactoring

Goal: Extract a piece of code into a separate function
Challenge: Computing input and output values of the code fragment

The extract method refactoring is one of the most useful and widely used code transformations. Many Java-oriented IDEs provide its semi-automated implementation with a variety of options. The extract method refactoring is well-studied in recent research papers, and correctness proofs have been provided [12, 15]. Adding closures to the language, however, brings new challenges to the implementation of this refactoring, especially when closures are allowed to reassign captured local variables.

One key aspect of the extract method refactoring is that it is almost exclusively about putting side-effects into the specific method. In Java all the side effects to the original method's local variables are arranged as an output values. All former local variables, which are input values of the analyzed piece of code, in turn become parameters of the newly introduced method. If the piece of code under consideration does not reassign any local variables and its resulting expression is not used, it is treated as a method with **`void`** return type. In other words, it deals only with side effects.

4.1 Scoping extracted functions

A piece of code that is going to be put into a separate method normally accesses two types of references: global and local. Both types of references are defined with respect to some scope. For example, in Java methods parameters and local variables are *locals*, whereas class fields are (relatively) *globals*. In Scala it is slightly different as functions may be nested and extracting a piece of code into a separate function requires that a scope for the newly introduced function be determined.

In the code below, we want to extract the grayed piece of code into a new function called `fun`. Before doing so, we need to decide where to put the new definition.

```
class A {
  def foo(i: Int) = {
    val j = i * 2

    def bar(k: Int) = {
      val r = k + i + j
      println(r)
    }

    println(bar(42))
  }
}
```

If the class `A` is the container of `fun`, then the function knows nothing about variables `i` and `j`, so they should be put into its signature as parameters. Conversely, if the new function is put inside of the body of function `foo`, these variables will already be in its scope, so the new function takes only one parameter, namely `k`. Finally, if it is put inside `bar`'s body, the new function does not need any parameters and it may be treated simply as a *lazily evaluated* expression.

4.2 Closures with state

In Scala local variables may be either immutable or mutable. There are no problems with the first type, but mutable variables in association with closures may give some interesting effects that must be treated correctly by the refactoring. For instance, the Scala code in the Figure 6 describes a function `foo` that returns a closure `cl`. This closure captures the outer mutable variable `a`, which is incremented every time the closure is called.

```
def foo = {
  var a = 239
  val cl = (i: Int) => {
    a = i + a
    println(a)
  }
  println(a)
  cl
}
```

Figure 6. Closure with state variable `a`

```
def myMethod(_a: Int) = {
  var a = _a
  (i: Int) => {
    a = i + a
    println(a)
  }
}
```

Figure 7. Extracted method returns closure with state

```
def foo = {
  var a = 42
  val cl = (i: Int) => {
   a = a + i
  }
  doSomething(cl)
  print(a)
}
```

Figure 8. Closure mutating local variable

The result is a closure with a strange side-effect, which might be useful for such techniques as memoization [2]. If one wants to extract the closure `cl` as a new method, the variable `a` must be treated as its *input value* but not as an *output* one. The point is that the closure `cl` does not affect the value of `a` inside `foo`: its effect is *delayed*. However, we cannot make `a` just a parameter of the new method: it will be immutable and its reassignment will be impossible. So, the correct solution is to extract the following method `myMethod` as shown in the Figure 7. The parameter `_a` serves as an initialization value for the inner "memoization variable" `a` of the returned closure.

Going further, consider Figure 8, which describes the definition of a closure inside of a method, followed by its subsequent usage. To perform the analysis for output variables of the selected fragment one must determine whether the closure `cl` is *invoked* by the function `doSomething`. Moreover, it may be invoked in the same execution thread immediately or asynchronously in another one. In the second case, the actual value of `a` may no longer matter to `foo`.

If nevertheless the value of `a` does matter, we should *wrap* it somehow to be able to refer to it after `doSomething` has been invoked on `cl`. One possible solution is to introduce an auxiliary class, assign `a` to its field and mutate these fields afterwards.[2] Using this implementation technique we do not care about the relative order of execution of `cl`'s body and the `print(a)` statement. The code on Figure 9 shows the result of applying the extract method refactoring to the code from Figure 8.

[2] This is actually what the Scala compiler does while processing closures.

```
class MyMethodEnv(var a: Int)

def myMethod1(env: MyMethodEnv) =
  (i: Int) => {
    env.a = env.a + i
  }

def foo = {
  val env = new MyMethodEnv(a = 42)
  val cl = myMethod1(env)
  doSomething(cl)
  println(env.a)
}
```

Figure 9. Refactored closure with an environment

The technique of using an auxiliary *environment* object is applied if the selected fragment has more than one output parameter. The necessary return values are packaged up into a wrapper object, which is returned and unwrapped again in the calling method.

5. More refactorings

In this section we give a short survey of other useful Scala-specific refactorings.

5.1 Splitting function parameters

In the Haskell programming language every function with more than one parameter may be partially applied or *curried*. To do the same with Scala functions, one should divide the list of parameters to several clauses. This is a modification of the well-known *change function signature* refactoring, which may be done automatically as shown in the listing below.

```
def sum(i: Int, j: Int) = i + j
println(sum(i,j))
```

⇓

```
def sum(i: Int)(j: Int) = i + j
println( sum(i)(j) )
```

A benefit of such a transformation is that one can now apply the *introduce variable* refactoring to the grayed code fragment to get a partially applied function of one parameter.

5.2 *Monadify* code refactoring

When moving to Scala from Java, programmers habitually write their code in an imperative style even in cases when it is not actually necessary. For example, computations with *nullable* results or breakable iterations through a list may be replaced by more concise and expressive *for-comprehensions*, using the appropriate monads from the standard Scala library. Figure 10 gives an example of this code transformation.

The key idea in detecting such patterns is a control-flow analysis that reveals exit conditions and default values (in the example such a default value is `null`).

5.3 *Structurize* refactoring

Structural subtyping is very helpful in object-oriented programs, as it makes them more generic and, as a consequence, more reusable [6]. For example, it helps to generalize the types of parameters of a function that cares only about the parameters' *structure*

```
def readValuePrice(fileName: String) =
  open(fileName) match {
  case Some(f) => readLine(f) match {
    case Some(key) =>
      ourDatabase.get(key) match {
        case Some(value) => getPrice(value)
        case None => null
    }
    case None => null
  }
  case _ => null
}
```

```
def readValuePrice(fileName: String) =
  for {f <- open(fileName)
       key <- readLine(f)
       value <- ourDatabase.get(key)}
       yield getPrice(value)
```

Figure 10. *Monadify* code refactoring

not their actual nominal types. It might be useful to automatically extract only those parts of a parameter's interface that matter for the given function, and replace the nominal type by a structural one. Figure 11 shows an example of this refactoring.

```
def getTailIfLong[T](i: Int, s: Seq[T]) =
  if (i > 0 && s. length >= i) s. tail
  else Nil
```

```
def getTailIfLong[T](i: Int,
    s: {def length: Int; def tail: Seq[T]}) =
  if (i > 0 && s.length >= i) s.tail
  else Nil
```

Figure 11. *Structurize* parameter type refactoring

Notice that we reified the abstract return type of the `tail` function with the exact type `Seq[T]`.[3] A subsequent step may extract the resulting structural parameter type as a type alias or as a separate trait.

6. Related work

Thompson et al. have worked on refactoring for functional languages for Haskell and Erlang in the HaRe and Wrangler programming tools, respectively [4]. Mechanical verification of refactorings for Haskell programs are outlined by Sultana and Thompson [14], who introduce the notion of type-based refactorings. For example, the *enlarge definition type* refactoring is a type-based refactoring that transforms a definition of a certain type into a coproduct with the original term as a left injection. The necessary pre- and post-conditions for this refactoring are stated and checked for correctness. *Tidier* is another tool for refactoring Erlang projects. It finds predefined code patterns and transform them automatically or interactively into more Erlang-specific constructs [10].

Schäfer et al. provide an extensible framework based on attribute grammars to implement sound *rename refactoring* for Java [11]. The implementation of *rename refactoring* in Scala in the IntelliJ plugin is not covered here, but its main idea is the same: all references are resolved beforehand and the mapping from declarations to usages is cached. There are some specific issues of renaming in Scala. For example, we should take into account the semantics of "special" methods, such as `unapply()` for extractors or `foreach()` for monad-like classes.

Code duplication detection is another domain closely related to program refactoring. The most common technique to find duplicates is a token-level approach that relies on suffix tree analysis. Many tools work also with annotates abstract syntactic trees (AST) to find clones up to α-conversion. The work [5] gives a detailed survey of different approaches for clone detection. It would be interesting to find *high-order* code duplicates, say, structurally recursive functions that are isomorphic up to the structure of *generalized algebraic data types*, which may be represented in Scala using *sealed* case classes.

Despite the fact that many classical patterns from object-oriented programming may be easer and much more concisely expressed in terms of functional programming, to the best of our knowledge, there is no work describing a unified approach to this kind of equivalence. For example, usages of the `Iterator` pattern without side effects may be expressed in terms of higher-order functions `map` or `reduce` over collections.

k-CFA, logic- and type-based control-flow analyses look like very promising techniques to extract different properties from functional code. They were successfully applied to analyze Scheme programs [7, 13], and we hope that they may be applied also to detect higher-order duplicates.

7. Conclusion and Future Work

In this paper we presented a survey of Scala-aware refactorings implemented in the IntelliJ IDEA programming environment. We described some issues typical for refactorings in Scala programs resulting from language features such as implicit conversions, dependent types and closures.

Some ideas of possible refactorings for Scala program were discussed. We hope that this will inspire the community to give more feedback about typical procedures used to improve Scala code and to make it more generic and functional.

Acknowledgments

We would like to thank the anonymous reviewers for providing valuable comments.

References

[1] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.

[2] Richard A. Frost and Barbara Szydlowski. Memoizing purely functional top-down backtracking language processors. *Sci. Comput. Program.*, 27(3):263–288, 1996.

[3] JetBrains Inc., http://www.jetbrains.com/idea/. *IntelliJ IDEA*, 2001.

[4] Huiqing Li and Simon Thompson. Tool Support for Refactoring Functional Programs. In Danny Dig, Robert Fuhrer, and Ralph Johnson, editors, *Proceedings of the Second ACM SIGPLAN Workshop on Refactoring Tools*, page 4pp, Nashville, Tennessee, USA, October 2008.

[5] Huiqing Li and Simon Thompson. Clone Detection and Removal for Erlang/OTP within a Refactoring Environment. In *ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM'09)*, Savannah, Georgia, USA, January 2009.

[6] Donna Malayeri and Jonathan Aldrich. Is structural subtyping useful? An empirical study. In *ESOP '09: Proceedings of the 18th European Symposium on Programming Languages and Systems*, pages 95–111, Berlin, Heidelberg, 2009. Springer-Verlag.

[3] See the `scala.collection.TraversableLike` trait for its implementation.

[7] Matthew Might. *Environment Analysis of Higher-Order Languages*. PhD thesis, Georgia Institute of Technology, June 2007.

[8] Martin Odersky. The Scala Language Specification. Available from http://www.scala-lang.org/, 2009.

[9] William F. Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, Champaign, IL, USA, 1992.

[10] Konstantinos Sagonas and Thanassis Avgerinos. Automatic refactoring of Erlang programs. In *PPDP '09: Proceedings of the 11th ACM SIGPLAN conference on Principles and practice of declarative programming*, pages 13–24, New York, NY, USA, 2009. ACM.

[11] Max Schäfer, Torbjörn Ekman, and Oege de Moor. Sound and extensible renaming for Java. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 277–294, New York, NY, USA, 2008. ACM.

[12] Max Schäfer, Mathieu Verbaere, Torbjörn Ekman, and Oege de Moor. Stepping stones over the refactoring rubicon. In Sophia Drossopoulou, editor, *ECOOP 2009 – Object-Oriented Programming*, volume 5653 of *Lecture Notes in Computer Science*, pages 369–393. Springer, 2009.

[13] Olin Grigsby Shivers. *Control-flow analysis of higher-order languages of taming lambda*. PhD thesis, Pittsburgh, PA, USA, 1991.

[14] Nik Sultana and Simon Thompson. Mechanical verification of refactorings. In *PEPM '08: Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 51–60, New York, NY, USA, 2008. ACM.

[15] Mathieu Verbaere, Ran Ettinger, and Oege de Moor. Jungl: a scripting language for refactoring. In Dieter Rombach and Mary Lou Soffa, editors, *ICSE'06: Proceedings of the 28th International Conference on Software Engineering*, pages 172–181, New York, NY, USA, 2006. ACM Press.