MAJOR TECHNICAL PROJECT ON

IDENTIFYING REFACTORING OPPORTUNITIES THAT PROMOTE FUNCTIONAL DESIGN PATTERNS IN SCALA

MTP REPORT

to be submitted by

NAMRATA MALKANI

B17096

for the award of the degree

of

BACHELOR OF TECHNOLOGY IN

COMPUTER SCIENCE AND ENGINEERING

SCHOOL OF COMPUTING AND ELECTRICAL ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY MANDI

June 2021

Abstract

A large software system should not only be functional, but in order to give continuous value over time, the source code needs to be readable, maintainable, have expressive architecture and have low complexity. For a given programming language these goals can be achieved by programming idiomatically. Idiomatic programming means writing code in a more succinct way by using the complete expressive power of the language through language-specific features or paradigms or recurring constructs.

Scala is a well-known functional language, and the problem addressed in this project is refactoring Scala source code in a way such that the transformed program is more idiomatic and conforms better to a functional way of writing programs. So far, the refactoring tool has been able to successfully identify and replace various types of control-flow constructs such as loops with up to 20 higher-order functions and common sequence methods available in Scala. It can also identify and replace various types of conditional expressions with pattern-match expressions, another well-known and efficient idiomatic feature of Scala.

All this has been achieved by metaprogramming: program analysis. The developed tool is extendable and future work involves implementing 'currying' as a step towards the implementation of more functional patterns in Scala and adding benchmarks for code-quality check.

Keywords: *meta-programming, syntax tree analysis, quasiquotes, Scala, Scalameta, functional programming, idiomatic features, control flow constructs, pattern-matching, higher-order functions, currying*

Contents

Chapter 1

Introduction

Refactoring is a program transformation that restructures existing code without without changing its external behavior and is a key practice in popular software design movements, such as agile. It is done to improve the design, structure, and/or implementation of the software while preserving its functionality. Automatic identification of refactoring opportunities is vital is large softwares. When done meticulously, refactoring improves code readability and reduced complexity; these can improve the source code's maintainability and create a simpler, cleaner, or more expressive architecture [18]. The problem addressed in this project is refactoring Scala source code in a way such that the transformed program conforms better to a functional design pattern [19].

The core idea of approaching the solution is program syntax tree analysis [15] - tree node traversal and pattern-matching for identification of faulty code, and tree re-construction by using 'quasiquotes' [17] for generation of refactored code [8]. Figure 1.1 systematically shows the broad process.

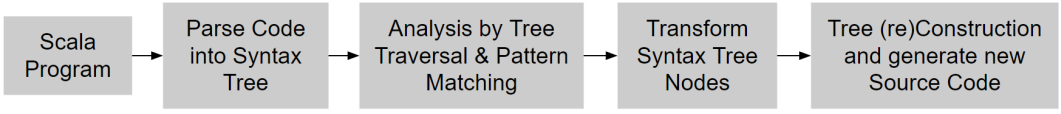

Figure 1.1: Schematic diagram of syntax tree analysis

This project is motivated by today's needs of software developers, especially Scala pro-

grammers. Code refactoring comes under the umbrella of problems like code quality, programmability, paradigm and structural shift to help developers write better programs - problems the industry finds quite useful. A large software system should not only be functional but in order to give continuous value over time, the source code needs to be readable, maintainable, have expressive architecture and have low complexity. For a given programming language these goals can be achieved by programming idiomatically. Idiomatic programming means writing code in a more succinct way by using the complete expressive power of the language through language-specific features or paradigms or recurring constructs. Scala is a well-known functional language, hence suggesting code improvements on code snippets (refactoring) which can be implemented with a functional design pattern and/or in an idiomatic manner that can improve efficiency, is the goal of this project.

Chapter 2

Background and Related Work

A code analysis and refactoring tool will heavily rely on a framework or library that can allow the user to read, generate and transform the underlying syntax tree of the source code. The guidelines and algorithm to identify and refactor a particular type of code are based on the motivation of refactoring- in this case, it is the promotion of functional design patterns in Scala.

2.1 Metaprogramming in Scala

Metaprogramming is a programming technique in which computer programs have the ability to treat other programs as their data. It means that a program can be designed to read, generate, analyze or transform other programs. Static program analysis, the kind used in this project, is based on metaprogram that does not modify itself while running. Proper code analysis and refactoring also allows programmers to minimize the number of lines of code to express a solution, in turn reducing development time. This is evident in tool performance in section 4.

In Scala, macros [13] have been traditionally used for code generation, static checking. They use Scala compiler's own ASTs. Since their release as an experimental feature of Scala 2.10, macros have brought previously impossible or prohibitively complex things to the realm of possible and are generally very tedious to use. For a long time, they were considered for this project, until more digging brought the Scalameta library [14] into focus. It is a

library to read, analyze, transform and generate Scala programs and is comparatively easier to use. Its wonderful user experience has been quite successful in rendering the macros less likely to be the choice of developers and has been promoted as a a complete replacement for them as well [15]. It's wide outreach and trusted performance can be gauged by the fact that it is a choice for Scala code review [5] by industry-wide used code quality checker tool like Codacy [4].

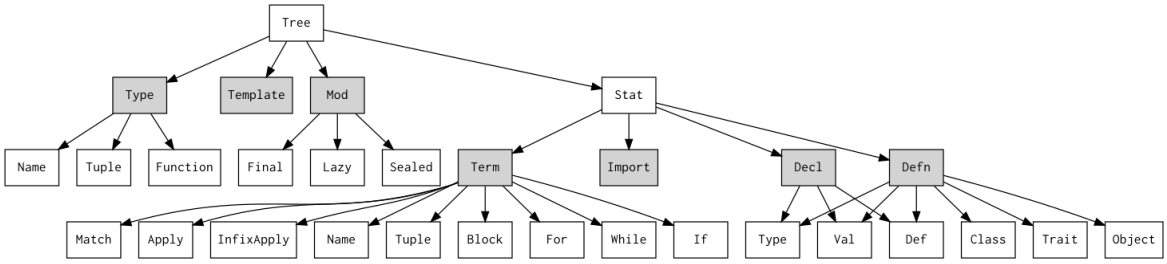

Figure 2.1: Scalameta has syntax trees that represent Scala programs.

A core functionality of Scalameta is syntax trees that it generates after analysing Scala code and enable developer to read, analyze, transform and generate Scala programs at a level of abstraction. Parsing, traversing, pattern matching and constructing syntax trees are key to refactoring techniques.

2.2 Functional Design Patterns

Functional world also has its own set of useful patterns. These patterns focus on writing code that avoids mutability and favors a declarative style, which helps us write simpler, more maintainable code. Sometimes these patterns have first-class language support. Functional patterns are more lightweight than object-oriented patterns. The implementation becomes as simple as a few lines of code. For many problems we encounter in programming, this style lets us work at a higher level of abstraction.

Functional languages give us a concise way of passing around a bit of computation without having to create a new class. Also, using expressions rather than statements lets us eliminate extraneous variables, and the declarative nature of many functional solutions lets us do in a single line of code what might take five lines in the imperative style. Some

object-oriented patterns can even be replaced with a straightforward application of functional language features [19].

Scala as a hybrid language:

From functional programming, Scala has absorbed functions as first class values and embraces the idea of immutability with various language constructs. Scala even supports lazy evaluation through call-by-name parameters and the lazy modifier for values. A combination from both object-oriented and functional worlds can be seen in Scala's ability to use pattern matching to deconstruct objects while still preserving encapsulation [7].

Scala has a rich collection of common sequence methods [6]. Developers prefer built-in methods for the common-sense reasons that they are less work to implement and are less error prone. Many functional language features and techniques of Scala have a similar effect on coding projects. While they may not be the exact equivalent to a pattern, they often give programmers techniques that are built-in alternative, less work and often produce code that is more concise and easier to understand than the original.

What Is Functional Programming?

At its core, functional programming is about immutability and about composing functions rather than objects [2].

Functional programs do the following:

- **Have first-class functions:** First-class functions are functions that can be passed around, dynamically created, stored in data structures, and treated like any other first-class object in the language.
- **Favor pure functions:** Pure functions are functions that have no side effects. A side effect is an action that the function does that modifies state outside the function.
- **Compose functions:** Functional programming favors building programs from the bottom up by composing functions together

- **Use expressions:** Functional programming favors expressions over statements. Expressions yield values. Statements do not and exist only to control the flow of a program.

- **Use Immutability:** Since functional programming favors pure functions, which cannot mutate data, it also makes heavy use of immutable data. Instead of modifying an existing data structure, a new one is efficiently created.

- **Transform, rather than mutate data:** Functional programming uses functions to transform immutable data. One data structure is put into the function, and a new immutable data structure comes out. This is in explicit contrast with the popular object-oriented model, which views objects as little packets of mutable state and behavior. A focus on immutable data leads to programs that are written in a more declarative style, since we cannot modify a data structure piece by piece.

Chapter 3

Scala Refactoring Tool

The research and groundwork laid to approach the problem and the motivation to follow with the proposed solution, have been discussed in 3.1. The design and implementation details have been presented in a comprehensive manner in section 3.2 and section 3.3.

3.1 Research and Groundwork

The initial work put in, to realize this project has been extensive. The choice of tools has been stated with explanations. Section 3.1 focuses on the preliminary research and skill acquisition. The choice of refactorings have been explained as well.

Language and Tools

- **Scala:** The project was implemented in and for Scala. It combines object-oriented and functional paradigms well and has a lightweight syntax, to define anonymous tasks easily (lambdas). It supports first-class citizenship of functions, allows nested functions, and supports curry [12].

- **Scalameta:** The foundation library for meta programming in Scala with a powerful parser for Scala code. It is industry-wide employed to explore and manipulate Scala code structurally and is an excellent choice for a static analysis tool [14].

- **IntelliJ IDEA:** Chosen as the IDE for development and is the ideal platform for plu-

gin deployment, given its exceptional usability due to being open-source and popular among Scala developers [9].

Preceding Studies

- **Courses:** CS-302 Paradigms of Programming and CS-502 Compiler Design, both offered by Dr. Manas Thakur helped strengthen the concepts and usage of various programming paradigms such as object-oriented, functional, logical, etc., and the usage of Abstract Syntax Trees for program analysis and transformation.

- **Research Papers:** Studying previous works like 'Crossing the Gap from Imperative to Functional Programming through Refactoring' [8], and 'Identifying Refactoring Opportunities for Replacing Type Code with Subclass and State' [18] ensured the utility of code restructuring and the adoption of tree manipulation techniques for the same.

- **Online Conferences:** Scala Meta Live Coding Session at *Scala Days Conferences* [3], Building IntelliJ IDEA plugins in Scala at *Scala in the City Conference* [16].

- **Tutorials and Useful Books:** Scala Tutorial(s) [11], [1], [7] Functional Design and Programming [2] IntelliJ-plugin development series [10].

Motivation

Common Sequence Methods:

In broad sense, the project is about identifying and refactoring as many loops as possible. Replacing a multi-line code snippet with a single line sequence method or higher-order function applied directly to an immutable data sequence, is in alignment with our motivation for the project - more declarative, more functional and more idiomatic style of programs.

Pattern Matching:

Pattern matching is a popular idiomatic feature of Scala and is much better and more efficient than the switch statements of object oriented counterparts like Java and C. But it is a good replacement for a variety of conditional statements. The reasons to decide between

if-else and pattern-matching has traditionally been focused on improving readability, and programming idiomatically. But there are multiple other reasons to consider it-

- It enforces a common return value and type for each of your branches.
- In languages with exhaustiveness checks (like Scala), it forces programmer to explicitly consider all cases
- It prevents early returns, which become harder to reason if they cascade, or the branches grow longer.
- It's 2020, the Scala compiler generates far more efficient byte-code in the pattern matching case.

3.2 Tool Design

The refactoring tool has been designed in such a manner that it is not only easily extendable with more refactorings (in future), but can also be imported in parts- based on the need of the type of refactoring, in other Scala projects.

Two main categories of control-flow constructs, often encountered in imperative and object-oriented programming, have been targeted in this version, to be refactored into functional and idiomatic style of Scala programs, while providing the same functionality in the source code.

- **Loops:** Various types of 'for loop' and 'while loops' have been analyzed thoroughly through a series of tree node pattern-matching and successive categorization 3.3, to be refactored into various common sequence methods and higher-order functions available in Scala. The package
- **Conditionals:** Algorithms to identify and refactor If-else statements and nested else-if statements into pattern match expressions have been properly implemented. Also, certain else-if statements enclosed within a loop can be also be refactored into a higher-order function that takes a pattern match expression as an anonymous function.

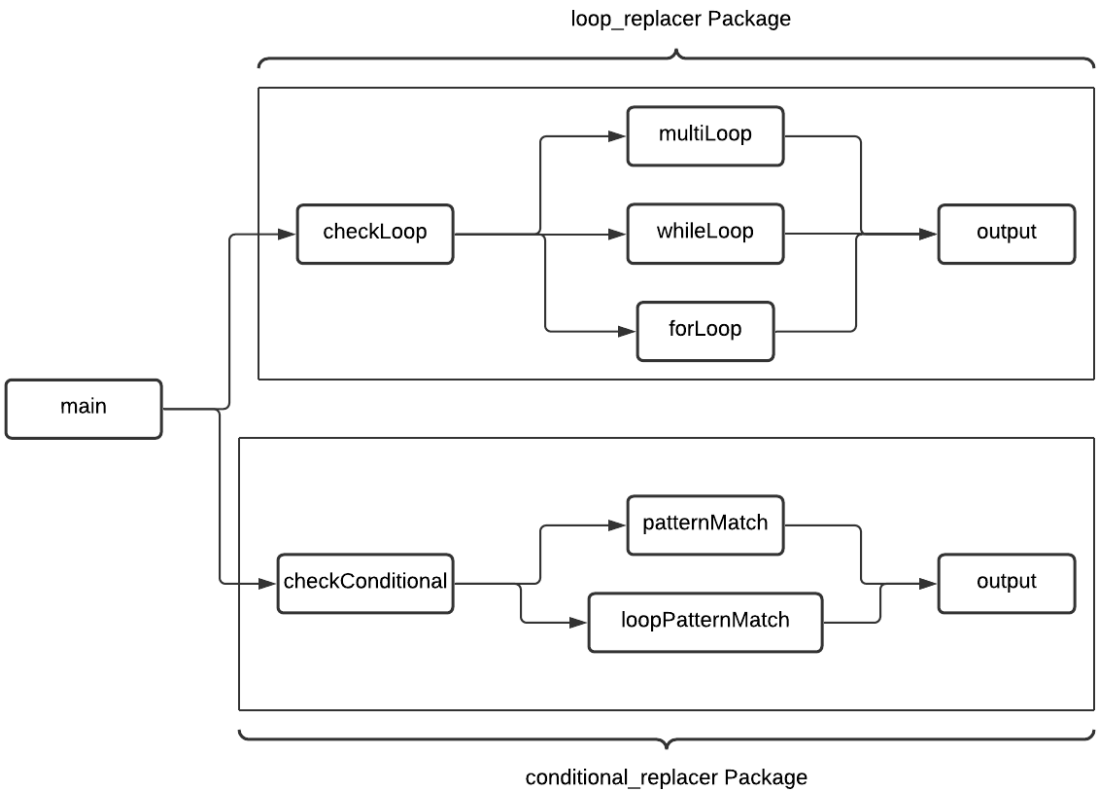

Figure 3.1: Design Overview of the Tool

As can be seen in Figure 3.1, the singleton Scala object file *main* is an entry point. It takes user program through a source code file or an IDE window, based on user choice and parses the program into a Scalameta syntax tree. It then invokes the *loop_replacer* package or the *conditional_replacer* package to analyze the tree and give the output as refactored code. The two packages can be individually imported in a Scala project and used directly as well.

Each singleton object file inside a package has certain definitions that perform key tasks involved in the refactoring algorithm. All the implementation details have been discussed in the next section.

3.3 Implementation

To simplify the reader's understanding of the tool implementation, the entire functional pipeline has been broken down into pieces, based on the singleton object encapsulating a group of functions. The functions- parameters, output and roles- have been explained, as well as how they interacts with each other.

To understand the implementation of loop-based refactorings in section 3.3.1, the above method and the flow chart shown in 3.3 would be enough. In section 3.3.2, the algorithm for rendering an if-else or nested else-if statement, into a pattern-match expression has been discussed in the form of a pseudo-code, in figure 3.8.

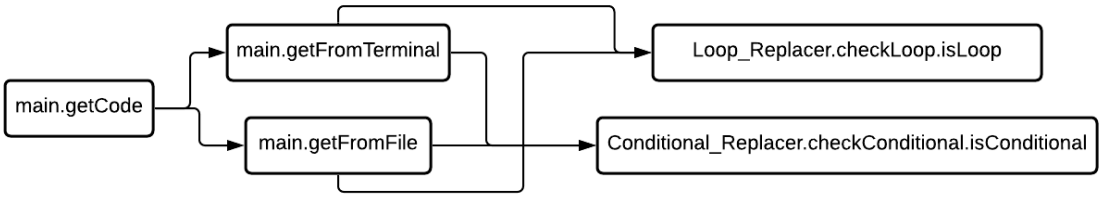

Figure 3.2: Definitions in the order of control flow: *main*

***main*:**

- `getCode()`: The function which is responsible for subsequent invocation of all that follows. Gives user choice between giving source code file as input or a code snippet in the IDE window.

- `getFromFile()`: Is called when user chooses file input. Reads and parses the code into a syntax tree and invokes `isLoop(tree)` of *checkLoop* and `isConditional(tree)` of *checkConditional*.

- `getFromTerminal()`: Is called when user chooses terminal input. Reads and parses the code into a syntax tree and invokes `isLoop(tree)` of *checkLoop* and `isConditional(tree)` of *checkConditional*.

3.3.1 Refactoring Control Flow Constructs: Loops

The main idea of the *loop_replacer* package is extracting the target code snippet (loop(s)) syntax tree from the source syntax tree, and after successive levels of analysis, categorizing it in the lowermost category, that brings us close to finding its ideal sequence method refactoring. This process flow can be understood in Figure 3.3. At the lowest stage of categorization, a loop or a couple of loops will be put into one of the 8 groups of 3-7 kinds of higher-order functions or common sequence methods. Further, finding and assigning one higher-order function or common sequence method as a refactoring to the said loop(s) is based on the tree-node analysis of the key assignment or conditional statement.

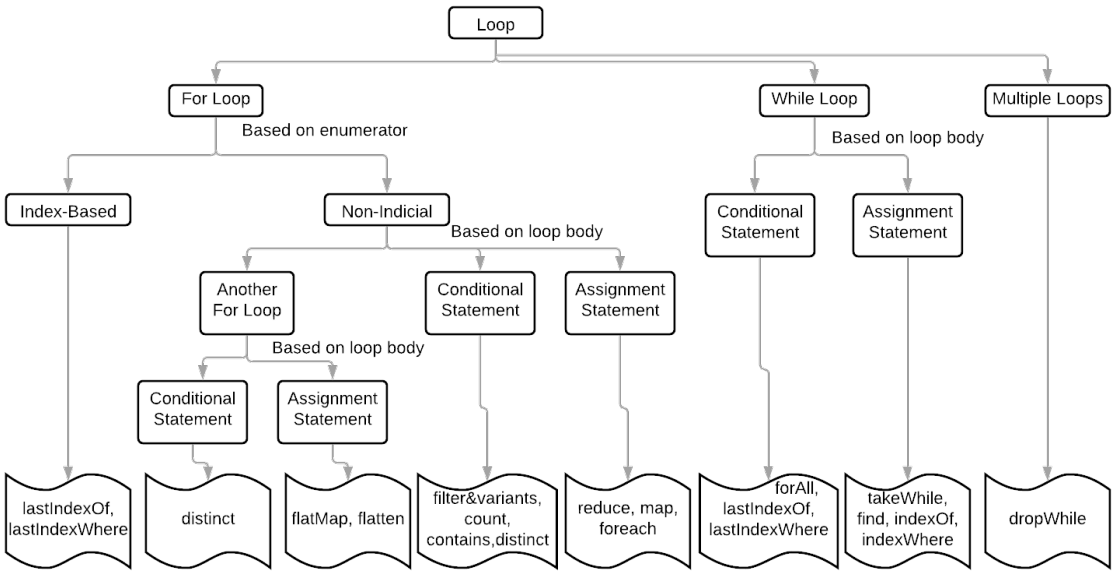

Figure 3.3: Categorization of loops corresponding possible refactorings

In Figures 3.4, 3.5, 3.6, and 3.7, the key functions, in the order of control flow have been shown and there is accompanied explanation of each.

checkLoop:

- `isLoop(tree)`: Traverses the program tree and checks for the presence of 'for' node or 'while' loop. If found, invokes `classifyLoop(tree)`.
- `classifyLoop(tree)`:Collects the 'for' nodes and 'while' nodes in separate Scala

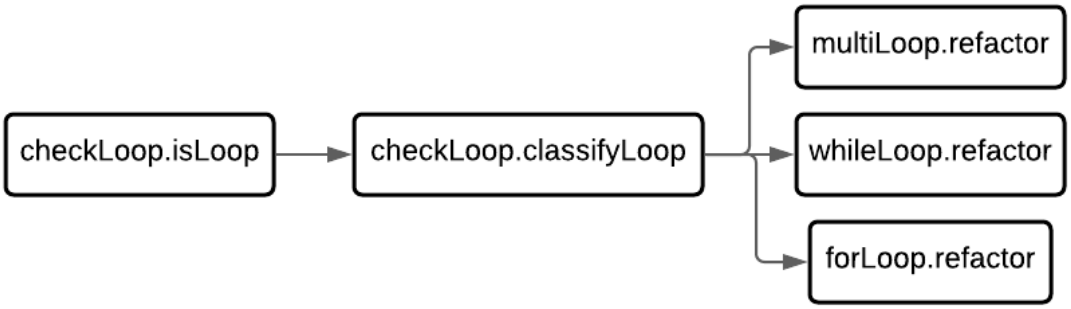

Figure 3.4: Definitions in the order of control flow: *checkLoop*

listBuffers: 'wLoops' and 'fLoops'. If a 'defn' node found, traverses it, if more than one loop found, collects all the loop nodes within, into a listBuffer: mLoops. Sends each of the lists to the `refactor(treeList)` of corresponding refactoring object: *forLoop*, *whileLoop*, *multiLoop*. `refactor(treeList)` in each case applies the *foreach* method on the list to invoke `classifyRefactoring(loopTree)`.

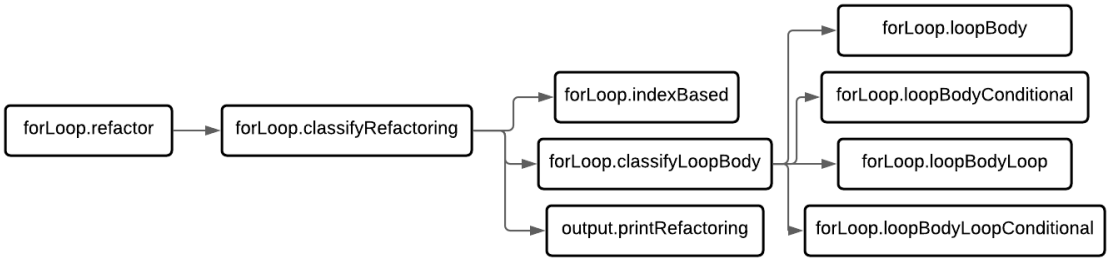

Figure 3.5: Definitions in the order of control flow: *forLoop*

forLoop:

`classifyRefactoring(loopTree)` checks the loop enumerator and invokes `classifyLoopBody(loopBodyNode):type` to classify the 'for' loop in one of the 5 main categories shown in 3.3. Subsequently the 5 functions corresponding to each type - `indexBased(loopEnumerator, loopBody)`, `loopBody(loopEnumerator, loopBody)`, `loopBodyConditional(loopEnumerator, loopBody)`,

`loopBodyLoop(loopEnumerator, loopBody)`, `loopBodyLoopCOnditional(loopEnumerator, loopBody)` - are invoked. To chose among the possible refactoring(s) of the category, extensive pattern matching on possible loop-body statement and/or conditional predicate and/or conditional clauses is done. If all the checks corresponding to a particular H.O. function or sequence method are passed, the refactored code is generated else, one or more 'possible' functional-replacers, based on partial fulfillment of the analysis, are presented to the user.

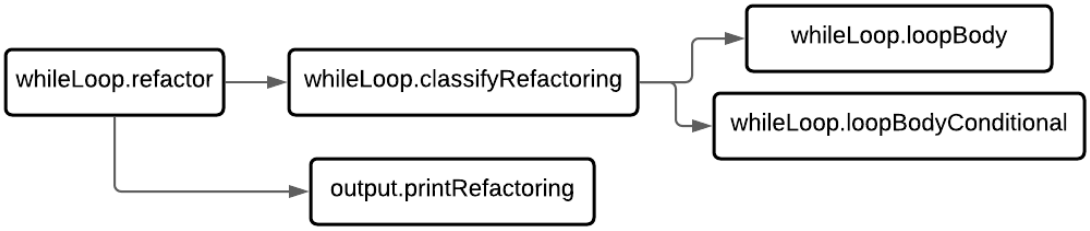

Figure 3.6: Definitions in the order of control flow: *whileLoop*

whileLoop:

`classifyRefactoring(loopTree)` categorizes the 'loop' in two broad categories and subsequent nailing down of the exact refactoring or alternative(s), is done in a manner similar as described for 'for' loops.

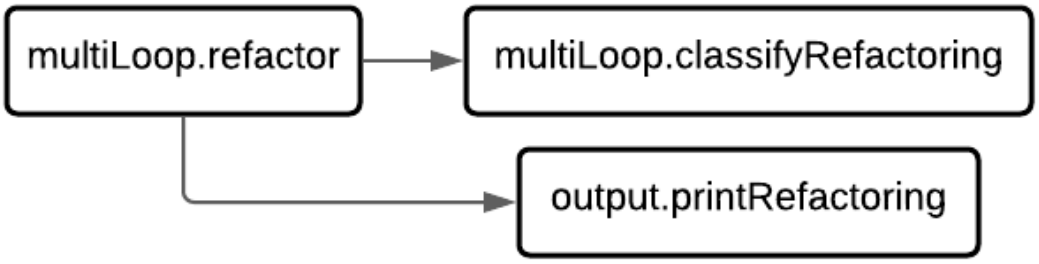

Figure 3.7: Definitions in the order of control flow: *multiLoop*

multiLoop:

In this case, `refactor(treeList)` applies *sliding(2).foreach* method on the list to invoke `classifyRefactoring((loopTree1,loopTree2)`. The two loops are analysed together through a series of node-quasiquote pattern-matching to check if they are working together to imperatively implement *dropwhile* method in Scala.

output:

The only function- `printRefactorings(originalTree, refactoredTree)` - prints the original code snippet with the refactored one, for comparison.

3.3.2 Refactoring Control Flow Constructs: Conditionals

The Figure 3.8 outlines the algorithm to convert a nested else-if statement into a pattern match. The first step is to identify a nested else-if statement, process it to store as a 'map' of predicates and statements, which will be used subsequently for constructing the pattern-match expression. This process is key to both *patternMatch* and *loopPatternMatch*.

The Figures 3.9 and 3.10 show the control flow of *conditional_replacer* package in parts, through the key functions.

checkConditional:

`isConditional(tree)` traverses the source code syntax tree and identifies the presence of a conditional or 'for loop' node. In case of a conditional node, it invokes `patternMatch.classifyRefactoring(node)` and in case of a loop node, it invokes `loopPatternMatch.classifyRefactoring(node)`.

patternMatch:

`classifyRefactoring(node)` produces the essential ¡if predicate-main clause¿ 'map' data structure from the conditional node and invokes `ifElseRefactor(map)`. It used the map for predicate analysis and generates the match expression or the 'tuple' set. If the

size of the tuple exceeds 3, the pattern match produces much worse code than the conditional and should be avoided. Otherwise `patternMatchExpression(map, tupleSet)` is invoked to construct the final pattern match refactoring. It invokes `modifyCaseClause(tupleSet)` on each predicate to get the case clauses inside the expression and plugs in the corresponding main clauses from the map directly.

***loopPatternMatch*:**

`classifyRefactoring(node)` invokes the `loopRefactor(loopBody, loopEnumerator)`. Here the loop body is traversed and the predicate-main clause map structure is produced and passed forward to the `predicateAnalysis(map, loopEnumerator)` function which analyses the predicates against the quasiquotes expressions referenced on the loop iterator, and transforms them to pattern match cases, when they fulfill the checks. The resultant match expression and new 'map' structure with cases and main clauses is returned. The final expression, enclosed in the Scala sequence method- map, applied on the original list in the conditional statements is produced. This particular refactoring only caters to the H.O function 'map'.

```
A nested else if statement:
If(A) w
Else if(B) x
Else y
Corresponding Scalameta tree node:
Term.if [
  Apply(A)  w
     Term.if [
          Apply(B)  x
               y]]]
Recursively traverse the tree to store the predicates and statements in
      a map:
{  A     -> w
   B     -> x
   else  -> y  }
Algorithm-
  Traverse the expression, match predicates and statements and store in
       a map data structure.
  2 types of cases based on size of map:
    A. If-else
       -> Simply plugin in the (only) predicate in match expression and
          have 2 cases: true/false. then plugin the statement in each.
    B. If-else-if-else-if-...-else
       -> Multiple cases based on predicate variety.
       -> Typical variety of predicates (boolean statements) considered-
          A==B  AB   A&&B  A!=B  A<B  A>B
       -> Analyze all predicates together and fill the set™ with tuple
          elements (to be used in match expression).
       -> Based on the size of tuple, identify the general case clause.
          Case clause for a tuple of 3 - (_,_,_).
       -> Analyze every predicates individually and return the modified
          case clause.
       -> Simply plugin the corresponding statement stored in the map.
```

Figure 3.8: Refactoring a conditional into a pattern match expression

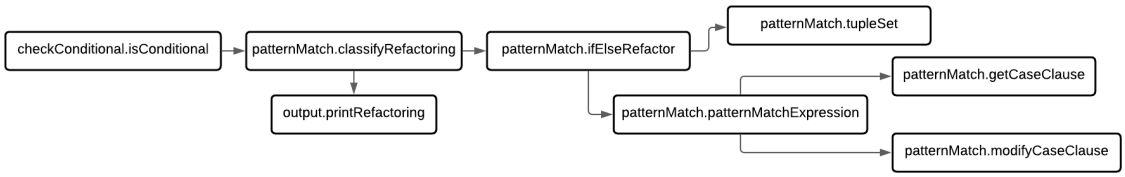

Figure 3.9: Definitions in the order of control flow: *patternMatch*

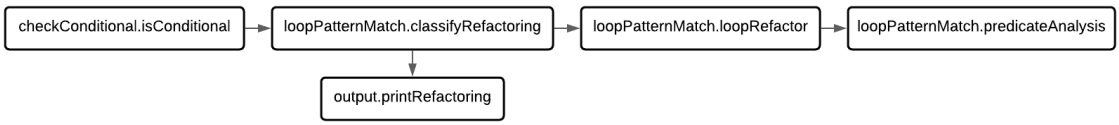

Figure 3.10: Definitions in the order of control flow: *loopPatternMatch*

Chapter 4

Tool Performance

The tool has been able to give exact refactored code and/or sequence method(s) that can be likely replacements, for a variety of ‘for’ loops and ‘while’ loops, from across 20 common sequence methods or higher-order functions available with Scala 2. In all the cases, a multi-line loop has been replaced by a single line sequence method, applied on some immutable Scala sequence or list.

The tool can also refactor almost any kind of nested else-if conditional statement into a pattern matching expression, unless the ‘match’ expression, against which all the case clauses will be checked, has to be a tuple of size greater than 4. In that case the tool lets the user know that converting the conditional into a pattern matching, will likely defeat the purpose of this refactoring - improving code readability. Such a pattern match expression is also likely to be much less efficient than the conditional and will have a ‘pattern guard’ - a required ‘if’ expression following a case clause.

A special case of conditional expression enclosed with a loop, refactor-able to a combination of sequence method ‘map’ and pattern match expression is also covered in conditional refactorings A2.7.

For a list of possible inputs and tool outputs - check A1 for loop-based refactorings and A2 for conditional refactorings.

Chapter 5

Conclusion and Future Work

The objective of this project was to create a tool that can help programmers produce better functional programs in Scala from existing code. As the first step towards this, commonly occurring imperative programming constructs like *loops and* conditionals were identified, categorized and efficient algorithms were designed to refactor them into better, succint functional code(s).

At 1300+ lines of code, the tool is composed of Scala packages - each takes care of a kind of refactoring and can be directly imported into other Scala projects. It is IDE independent, extendable, and makes adding new refactorings easy.

Thus, one of the future goals of the project, is adding more refactorings- now tackling even larger OO patterns and refactoring them into a possible functional counterpart. It is possible to implement OO patterns and principles using functional techniques. For example, decorator pattern can be implemented by functional composition (creating a functional composition pipe with the help of currying). As the first step towards this goal, implementing an algorithm for refactoring a function to apply *currying* or *partial application*, would be ideal.

In order to get the full advantage of the tool, it is also necessary to evaluate the original and refactored program with respect to a code evaluation metric and compare performances. Hence designing a code-quality check system will also be a key part of future work.

Appendices

Appendix A1

The tables below show the results related to the *loop_replacer* package of the tool.

Tool Output	Loop code snippet

xs.reduce((x, y) =⟩ x + y)	`for (x <- xs) {`
	`  sum += x`
	`}`

xs.distinct	`for(x <- xs){`
	`  if(!l.contains(x))`
	`  l += x`
	`}`

xs.filter(h).map(x =⟩ g(x))	`for (x <- xs) {`
	`  if (h(x)) list += g(x)`
	`}`

xs.foreach(println)	`for (x <- xs) {`
	`  println(x)`
	`}`

xs.lastIndexOf(x)	`for(i <- xs.indices){`
	`  if(xs(i)==x)`
	`  index=i`
	`}`

xs.count(_ = x)	`for(x <- xs){`
	`  if(x==t)`
	`  count=count+1`
	`}`

xs.forall(h)	`while(ans){`
	`  if(!h(xs(i)))`
	`  ans=false`
	`  i = i+1`
	`}`

xs.lastIndexWhere(x =⟩ h(x))	`for(i <- xs.indices){`
	`  if(h(xs(i)))`
	`  index=i`
	`}`

Table A1.1: Refactored code output and original code snippet

Tool Output	Loop code snippet

xs.map(x =⟩ g(x))	`for (x <- xs) {`
	`  list += g(x) //2*x`
	`}`

xs.filter(h)	`for (x <- xs) {`
	`  if (h(x)) list += g(x)`
	`}`

xs.takeWhile(h)	`while (h(xs(i))) {`
	`  list += xs(i)`
	`  i +=1`
	`}`

xs.filterNot(h)	`for (x <- xs) {`
	`  if (!h(x)) list += x`
	`}`

xs.indexOf(x)	`while(index!=xs.length && xs(index)!=x){`
	`  index = index+1 //index += 1`
	`}`

xs.filter(h).foreach(println)	`for (x <- xs) {`
	`  if(h(x))`
	`  println(x)`
	`}`

xs.contains(x)	`for(x <- xs){`
	`  if(x==t)`
	`  present=true`
	`}`

Table A1.2: Refactored code output and original code snippet

Tool Output	Loop code snippet

Likely sequence method- flatten	`for(x <- xs){`
	`  for(i <- x){`
	`  ans += i`
	`  }`
	`}`

Likely sequence method- flatMap	`for(x <- xs){`
	`  for(t <- f(x)){`
	`  ans += t`
	`  }`
	`}`

Table A1.3: Likely refactorings detected and original code snippet

Tool Output	Loop code snippet

$xs.indexWhere(x => h(x))$
Likely Sequence Methods:indexWhere, find
	`while(i<xs.length && !h(xs(i)) ){`
	`  i = i+1`
	`}`

$loop1 : xs.takeWhile(h)$
$loop2 : Norefactoringfound$
xs.dropWhile(h)
	`while (h(xs(i))) {`
	`  i +=1`
	`}`
	`while(i!=xs.length){`
	`  list += xs(i)`
	`  i+=1`
	`}`

$xs.indexWhere(x => h(x))$
Likely Sequence Methods:indexWhere, find
	`while(index<xs.length && !h(xs(index))){`
	`  index = index+1`
	`}`

Table A1.4: Refactored code output and additional possible refactorings detected and original code snippet

Appendix A2

The figures here, show the input conditional code snippet and resultant output. These are results related to the *conditional_replacer* package functionalities.

```
1 if (password == enteredPassword) {
2    println(s"User is authenticated.")
3 }
4 else {
5    println(s"Entered password is invalid.")
6 }
```

Figure A2.1: conditional code snippet-1

```
1 password == enteredPassword match {
2 case true => {
3    println(s"User is authenticated.")
4 }
5 case false => {
6    println(s"Entered password is invalid.")
7 }
8 }
```

Figure A2.2: Output for code snippet-1

```
if(a==100){
    val d =1
    println("100")
}
else if(a==200){
    val d=2
    println("2000")
}
else {
    val d=3
    println("none")
}
```

Figure A2.3: conditional code snippet-2

```
a match {
case 100 => {
    val d =1
    println("100")
}
case 200 => {
    val d=2
    println("2000")
}
case _ => {
    val d=3
    println("none")
}
```

Figure A2.4: Output for code snippet-2

```
if(a == 0) f(a)
else if(b == 4) g(b)
else if(c > 90) f(c)
else if(c > 10 && a == 20) h(a,c)
else a + b + c
```

Figure A2.5: conditional code snippet-3

```
(a, b) match {
case (0,_) => f(a)
case (_,4) => g(b)
case (_,_) if(c > 90) => f(c)
case (_,_) if(c > 10 && a == 20) => h(a,c)
case (_,_) => a + b + c
}
```

Figure A2.6: Output for code snippet-3

```
for (x <- xs){
    if(x==1) result += f(x+1)
    else if(x==2) result += f(x+2)
    else if(x==3) result += f(x+3)
    else result += f(x+4)
  }
```

Figure A2.7: conditional code snippet-4

```
1 result = xs.map{
2      x => x match {
3         case 1 => f(x+1)
4         case 2 => f(x+2)
5         case 3 => f(x+3)
6         case _ => f(x+4)
7       }
8    }
```

Figure A2.8: Output for code snippet-4

Bibliography

[1] Derek Banas. *Scala Tutorial*. https://www.youtube.com/watch?v=DzFt0YkZo8M&list=PLwbCdQeXHivR3IEN6cwxPAByKUkQ3Tqrp&index=19. Accessed: September 2020.

[2] Michael Bevilacqua-Linn. *Functional Programming Patterns. in Scala and Clojure*. The Pragmatic Programmers, LLC., 2013.

[3] Pathikrit Bhowmick. *Scala Meta Live Coding Session Scala Days Conferences*. https://www.youtube.com/watch?v=tXWBx2yVIEQ&t=1s. Accessed: September 2020.

[4] *Codacy*. https://www.codacy.com/. Cited: June 2021.

[5] *Codacy tool for Scalameta*. https://github.com/codacy/codacy-scalameta. Cited: June 2021.

[6] *Common Sequence Methods*. https://docs.scala-lang.org/overviews/scala-book/collections-methods.html#inner-main. Accessed: September 2020.

[7] Alex Payne Dean Wampler. *Programming Scala*. O'Rilley Media, Inc., 2014.

[8] Alex Gyori et al. "Crossing the gap from imperative to functional programming through refactoring". In: (Aug. 2013). DOI: 10.1145/2491411.2491461.

[9] *IntelliJ Platform SDK*. https://plugins.jetbrains.com/docs/intellij/welcome.html. Accessed: September 2020.

[10] JetBrainsTV. *Busy plugin developers series.* https://www.youtube.com/watch?v=-6D5-xEaYig&list=PLwbCdQeXHivR3IEN6cwxPAByKUkQ3Tqrp&index=4&t=493s. Accessed: December 2020.

[11] Jordan Parmer. *Code Real World App Using Purely Functional Techniques (in Scala).* https://www.youtube.com/watch?v=m40YOZr1nxQ&t=1s. Accessed: October 2020.

[12] *Scala Documentation*. https://docs.scala-lang.org/. Accessed: September 2020.

[13] *Scala Macros*. https://docs.scala-lang.org/overviews/macros/paradise.html#inner-main. Cited: June 2021.

[14] *Scalameta · Library to read, analyze, transform and generate Scala programs*. https://scalameta.org/. Accessed: October 2020.

[15] Devon Stewart. *Automate More Code With Scalameta C 2019 Conference*. https://www.youtube.com/watch?v=_gk2-1kO-l0. Accessed: October 2020.

[16] Igal Tabachnik. *Building IntelliJ IDEA plugins Scala in the City Conference*. https://www.youtube.com/watch?v=IPO-cY_giNA&list=PLwbCdQeXHivR3IEN6cwxPAByK
index=7&t=9s. Accessed: January 2020.

[17] *tree/quasiquotes guide*. https://scalameta.org/docs/trees/quasiquotes.html. Cited: June 2021.

[18] Jyothi Vedurada and V Krishna Nandivada. "Refactoring Opportunities for Replacing Type Code with State and Subclass". In: *Proceedings of the 39th International Conference on Software Engineering Companion*. ICSE-C '17. Buenos Aires, Argentina: IEEE Press, 2017, 305"307. ISBN: 9781538615898. DOI: 10.1109/ICSE-C.2017.97. URL: https://doi.org/10.1109/ICSE-C.2017.97.

[19] Scott Wlaschin. *Functional programming design patterns*. https://fsharpforfunandprofit.com/fppatterns/. Accessed: November 2020.