MAJOR TECHNICAL PROJECT ON

IDENTIFYING REFACTORING OPPORTUNITIES THAT PROMOTE FUNCTIONAL DESIGN PATTERNS IN SCALA

INTERIM PROGRESS 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

February 2021

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 [12]. 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 [13].

The core idea of approaching the solution is program syntax tree analysis [10]. The two main steps towards achieving the desired result are tree node traversal for identification and node manipulation/transformation for new syntax [3]. Figure 1.1 systematically shows the broad process.

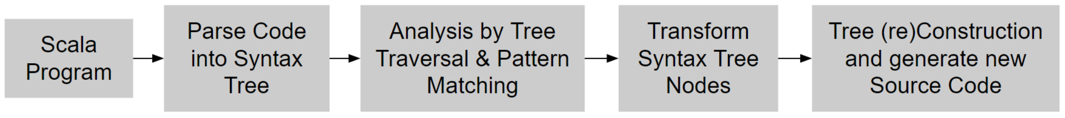
Scala Program
Parse Code into Syntax Tree
Analysis by Tree Traversal & Pattern Matching
Transform Syntax Tree Nodes
Tree (re)Construction and generate new Source Code

Figure 1.1: Schematic diagram of speech production.

This project is motivated by today's needs of software developers, especially Scala programmers. 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, 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/need to be implemented with a functional design pattern and assessing the transformed code on a code quality metric is the goal of this project.

Chapter 2

Objective and Scope of Work

The objective of this work is to produce better functional programs in Scala from existing code. As the first step towards this is choosing a commonly occuring programming construct, *loops*, that can be improved by inculcating elements of functional programming, *higher order functions*.

The above choice is supported by the fact that certain problems are better solved by adopting one particular programming paradigm (functional in this case) over another. Higher order functions (filter, map, reduce etc.) are mutable, easily parallelized, compared to 'loops' that may perform the same task.

Course of action:

- Identify the type of refactoring suitable for the body of a Scala function. The details of algorithm designed are found in Section 3.3.
- After successful identification of the category, transform the function body, and return new source code to the user. For implementation, see Section 3.4.

Current Limitations and scope for future:

- This particular problem is motivated by a leading healthcare-analytics company based at Chennai, hence it comes under the purview of real world software engineering issues.

- The current version is a command-line tool. It does not include code highlighting for informing the user, but a plugin based future version is a possibility.
- A limited number of refactorings have been implemented so far, currently only catering to *loop-to-higher-order-function*, more refactorings focusing on functional design patterns will be researched and included in the final version.
- In order to get the full advantage of the tool, it is also necessary to evaluate the refactored program with respect to a code evaluation metric and compare with the original code's performance on the same. Hence designing and assessing codes on this metric will be a key part of future work.
- Finally assessing the tool on a number of open-source large software systems will also be included in the coming versions.

Chapter 3

Work Done

All the research and final work done so far have been put into four sections. In Section 3.1 the choice of tools has been stated with explanations. Section 3.2 focuses on the preliminary research and skill acquisition. In Section 3.3 and Section 3.4, the algorithms for identification and implementation for the refactorings have been discussed.

3.1 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 [8].

- **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 [9].

- **IntelliJ IDEA:** Chosen as the IDE for development and is the ideal platform for plugin deployment, given its exceptional usability due to being open-source and popular among Scala developers [4].

- **JetBrains sbt-idea-plugin:** A Github repository maintained by JetBrains, to develop IntelliJ plugins with Scala and Scala Build Tool (SBT) [5].

3.2 Initial Research and Groundwork

- **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' [3], and 'Identifying Refactoring Opportunities for Replacing Type Code with Subclass and State' [12] 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* [2], Building IntelliJ IDEA plugins in Scala at *Scala in the City Conference* [11].
- **Tutorials:** Scala Tutorial(s) [7], [1], IntelliJ-plugin development series [6].

3.3 Progress Made: Identifying the Type of Refactoring

In the following Section, the design of each function written for identification of the type of refactoring and how the functions interact with each other to complete the algorithm has been shown through Figures 3.1, 3.2, 3.3 and 3.4.

The examples of input and output on implementation of this algorithm (*identifyRefactoring.scala*) are shown in Appendix A1.

It is worth noting that the pattern-matching of tree nodes required in the function `HigherOrderFunctionType` has been done with the help of Scalameta quasiquotes [9]. They are string interpolators that allow easy manipulation of Scala syntax trees by expanding at compile-time into normal tree constructor calls. Other than quasiquotes, custom tree traversals and transformations are the two most useful and powerful feature of the Scalameta parser.

The input program is parsed into a Scala syntax tree and fed to the

```
Identifier(programTree){
    Recursively visit programTree nodes;
    if(Dfn is a definition node)
        Refactoring = LoopBodyPatternType(Tree node of Defn body);
    return Refactoring;
}
```

Figure 3.1: Identifier: *identifyRefactoring.scala*

`Identify(programTree)` function. Through tree traversal, every definition node in the program tree is targeted. A definition node itself is composed of 6 tree nodes, one of them being the definition body node, which is given as input to `LoopBodyPatternType(DefBody)`. Once again through tree traversal and with the help of some helper functions, all components of any given loop node in the definition are found and analysed. To determine the exact nature of refactoring, the combination of the outputs of `HigherOrderFunctionType(LoopBody, LoopIterator)` and `AccumulatorDataType(DefBody, Accumulator)` is used.

3.4 Progress Made: Transforming the Program

In the following Section, the design of each function written for program transformation and how the functions interact with each other to complete the algorithm has been shown through Figures 3.5, 3.6, 3.7 and 3.8.

The examples of input and output code snippets on implementation of this algorithm (*refactor.scala*) are shown in Appendix A1.

The input program is parsed into a Scala syntax tree and any definition node found is fed to the `Refactor(DefnTree)` function. It analyses and if refactoring is possible, transforms the loop nodes found in the definition body tree by calling the `LoopRefactor(LoopNode, DefnTree)` function. This function extracts all the im-

```
LoopBodyPatternType(DefB){
  Recursively visit DefB nodes;
  if(L is a loop node){
    LBody = GetBodyOfLoopNode(L);
    LIterator = GetIteratorOfLoopNode(L); //'x' in for(x<-xs){LBody}
    RefactoringType = HigherOrderFunctionType(LBody, LIterator)->
      tuple1;
    Accumulator = HigherOrderFunctionType(LBody, LIterator)->tuple2;
    DataType = AccumulatorDataType(DefB, Accumulator);
    Based on the combination of RefactoringType & DataType:
      return "reduce";    or
      return "map";       or
      return "filter";    or
      return "filter & map"; or
      return "none";
  }
  return "none";
}
```

Figure 3.2: LoopBodyPatternType: *identifyRefactoring.scala*

portant elements of the loop and the loop body and calls for `HigherOrderFunctionBody(DefnTree, LoopBody, LoopIterator,OriginalList)`. If refactoring is possible, `HigherOrderFunctionBody` returns the refactored node and updates the loop body accumulator variable name globally. This is important later for the `Refactor(DefnTree)` function, as this function needs to transform the 'accumulator return' node and the 'accumulator define' nodes to ensure no change in the external behavior of the program.

```
HigherOrderFunctionType(LoopBody, LoopIterator){
   Recursively visit LoopBody nodes;
   node N match {
      case: Accumulator += LoopIterator => RefactorType =1;
      case: Accumulator += F(LoopIterator) => RefactorType =2;
      case: if(G(Accumulator)) Accumulator += LoopIterator =>
         RefactorType =3;
      case: if(G(Accumulator)) Accumulator += H(LoopIterator) =>
         RefactorType =4
   }
   return (RefactorType, Accumulator.toString)
}
```

Figure 3.3: HigherOrderFunctionType: *identifyRefactoring.scala*

```
AccumulatorDataType(DefBody, VarName){
   Recursively visit DefBody nodes;
   if(DefV is the VarName definition node){
      if(DefV defines Integer Literal)
         return "INT";
      if(DefV defines ListBuffer)
         return "List";
   }
   return EmptyString;
}
```

Figure 3.4: AccumulatorDataType: *identifyRefactoring.scala*

```
Refactor(DefnTree){
    Recursively visit Defn nodes;
     node N match {
        case: N is loop node => LoopRefactor(N, DefnTree)
        case: N is VarName 'ListBuffer' definition node => Varname 'List'
              definition node //Varname is global Variable: 'Accumulator'
             in the loop, updated by LoopRefactor() if refactoring found
        case: N is 'return VarName.toList' node => 'return VarName' node
     }
     return transformed DefnTree
}
```

Figure 3.5: Refactor: *refactor.scala*

```
LoopRefactor(LoopTree, DefnTree){
    LBody = GetBodyOfLoopNode(LoopTree);
    LIterator = GetIteratorOfLoopNode(LoopTree);
    OriginalList = GetOriginalListOfLoopNode(LoopTree); //'xs' in for(x
        <-xs){LBody}
    Refactored = HigherOrderFunctionBody(DefnTree, LBody, LIterator,
        OriginalList)->tuple1;
    RefactoredCode = HigherOrderFunctionBody(DefnTree, LBody, LIterator,
         OriginalList)->tuple2;
    if(Refactored)
        return RefactoredCode;
    else
        return LoopTree
}
```

Figure 3.6: LoopRefactor: *refactor.scala*

```
HigherOrderFunctionBody(DefnTree, LoopBodyTree, LoopIterator,
    OriginalList){
   Recursively visit LoopBodyTree nodes;
   node N match {
      case i: i <- 1 to 4 //similar but more intricate pattern matching
          technique as HigherOrderFunctionType() in
         identifyRefactoring.scala. New node is constructed as a
         quasiquote. The AccumulatorDataType() is called to ensure
         datatype of accumulator. If one of the cases match,
         Refactorable=true. And Accumulator VarName is also to be
         updated (global variable).
   }
   return (Refactorable, RefactoredCode)
}
```

Figure 3.7: HigherOrderFunctionBody: *refactor.scala*

```
AccumulatorDataType(DefB, VarName){
   Recursively visit DefB nodes;
   if(DefV is the VarName definition node){
      if(DefV defines Integer Literal)
         return "INT";
      if(DefV defines ListBuffer)
         return "List";
   }
   return EmptyString;
}
```

Figure 3.8: AccumulatorDataType: *refactor.scala*

Appendices

Appendix A1

The inputs and outputs of *identifyRefactoring.scala* and *refactor.scala*, which are the implementations of the algorithm discussed in Section 3.3 and Section 3.4 respectively, are given below.

Outputs (*identifyRefactoring.scala*):

- For input Fig. A1.1 - *reduce*
- For input Fig. A1.3 - *map*
- For input Fig. A1.5 - *filter*
- For input Fig. A1.7 - *filter and map*

```scala
def sumLoop(xs: List[Int]): Int = {
   var sum = 0
   for (x <- xs) {
      sum += x
   }
   return sum
}
```

Figure A1.1: Scala definition that returns the sum of a list

```
def sumLoop(xs: List[Int]): Int = {
   var sum = 0
   sum = xs.reduce((x, y) => x + y)
   return sum
}
```

Figure A1.2: Output of *refactor.scala* for input Fig. A1.1

```
def f(x: Int): Int = {
   x * 2
}
def mapLoop(xs: List[Int]): List[Int] = {
   var list = ListBuffer[Int]()
   for (x <- xs) {
      list += f(x)
   }
   list.toList
}
```

Figure A1.3: Scala definition that returns a new list by applying *f* to the input list

```
def f(x: Int): Int = {
   x * 2
}
def mapLoop(xs: List[Int]): List[Int] = {
   var list = List(0)
   list = xs.map { (x: Int) => f(x) }
   list
}
```

Figure A1.4: Output of *refactor.scala* for input Fig. A1.3

```
def filterLoop(xs: List[Int]): List[Int] = {
   var list = ListBuffer[Int]()
   for (x <- xs) {
      if (x % 2 == 0) list += x
   }
   list.toList
}
```

Figure A1.5: Scala definition that returns a new list by filtering the input list

```
def filterLoop(xs: List[Int]): List[Int] = {
   var list = List(0)
   list = xs.filter(_ % 2 == 0)
   list
}
```

Figure A1.6: Output of *refactor.scala* for input Fig. A1.5

```
def f(x: Int): Int = {
   x * 2
}
def mapFilterLoop(xs: List[Int]): List[Int] = {
   var list = ListBuffer[Int]()
   for (x <- xs) {
      if (x%2==0) list += f(x)
   }
   list.toList
}
```

Figure A1.7: Scala definition that returns a new list by applying *f* to the filtered input list

```
def f(x: Int): Int = {
   x * 2
}
def mapFilterLoop(xs: List[Int]): List[Int] = {
   var list = (xs: List[Int]) =>
   {
      var refactorVal1 = xs.filter(_ % 2 == 0)
      var refactorVal2 = refactorVal1.map { (x: Int) => f(x) }
      refactorVal2
   }
   list(xs)
}
```

Figure A1.8: Output of *refactor.scala* for input Fig. A1.7

Bibliography

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

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

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

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

[5] JetBrains. *SBT IDEA Plugin*. `https://github.com/JetBrains/sbt-idea-plugin`. Accessed: December 2020.

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

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

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

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

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

[11] 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.

[12] 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, pp. 305–307. ISBN: 9781538615898. DOI: `10.1109/ICSE-C.2017.97`. URL: `https://doi.org/10.1109/ICSE-C.2017.97`.

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