Refactoring Scala Programs to Promote Functional Design Patterns

Namrata Malkani
IIT Mandi, India
b17096@students.iitmandi.ac.in

Manas Thakur
IIT Mandi, India
manas@iitmandi.ac.in

1 Introduction

Refactoring is a program transformation that is done to improve the design, structure, and/or implementation of the software while preserving its functionality. Automatic identification of refactoring opportunities is vital in large software [3]. The problem addressed in this work is refactoring Scala source code in a way that the transformed program conforms better to a functional style of programming.

Scala is a well-known functional language. It has functions as first-class citizens and embraces the idea of immutability with various language constructs. Functional design patterns, generally, focus on writing code that avoids mutability and favors a declarative style. To give continuous value over time, the code needs to be readable and maintainable, and should have an expressive architecture with low complexity. For a given programming language, these goals can be achieved by programming idiomatically. This means writing code in a more succinct way by using the complete expressive power of the language. Developers also prefer these methods for the simple reason that they are less work to implement, less error prone, concise, and easier to understand. Many functional language features and techniques of Scala have a similar effect on coding projects [1].

Imperative patterns, such as loops that manipulate data, can be replaced with a straightforward application of Scala's functional features such as its rich collection of common sequence methods. The declarative nature of such functional solutions lets us do in a single line of code what might take multiple lines in the imperative style, improved readability, and is much less error prone as the functional method produces a new data list instead of mutating the original one.

Pattern matching, another popular idiomatic feature of Scala, is a good replacement for a variety of conditional statements. The reasons to decide between nested else-if statements and pattern-matching has traditionally been focused on improving readability. But there are multiple other reasons such as enforcing a common return value, forcing programmer to explicitly consider all the cases in a language with exhaustive checks like Scala, and generating more efficient byte code in newer versions of Scala 2.

Hence, in alignment with the motivation, the goal of this work is to suggest code improvements on code snippets (refactoring) that can be implemented in a functional or an idiomatic manner. The focus is on refactoring two commonly used control-flow constructs: loops and conditionals.

ISSTA' 21, July 14-15, 2021, Virtual Event
2021. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

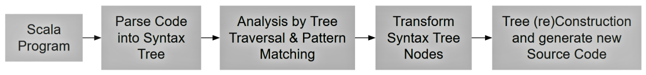
Scala Program
Parse Code into Syntax Tree
Analysis by Tree Traversal & Pattern Matching
Transform Syntax Tree Nodes
Tree (re)Construction and generate new Source Code

Figure 1. Schematic diagram of syntax tree analysis.

Static program analysis, which is vital to code refactoring, heavily relies on a framework or library that can allow the user to read, generate and transform the underlying syntax tree of the source code. In this work, we use the widely popular metaprogramming library Scalameta [2]. Parsing, traversing, pattern matching and constructing syntax trees are key to code analysis using Scalameta; see Figure 1.

2 ScalaRT: Tool Design

The refactoring framework 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 (individual packages), 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.

As can be seen in Figure 2, 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 and the *conditional_replacer* package to analyze the tree, and gives the output as refactored code. Each singleton object file within a package encapsulates a group of functions. Together, they compose the functional data-flow pipeline to implement the refactoring algorithm(s).

The main idea of the *loop_replacer* package is to extract 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 with Figure 3. At the lowest stage of categorization, a loop or a couple of loops will be put into one of the eight groups of 3-7 kinds of higher-order functions or

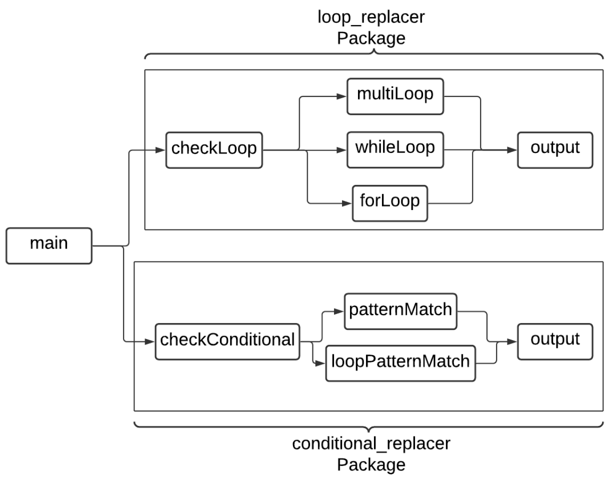

Figure 2. ScalaRT: Design overview.

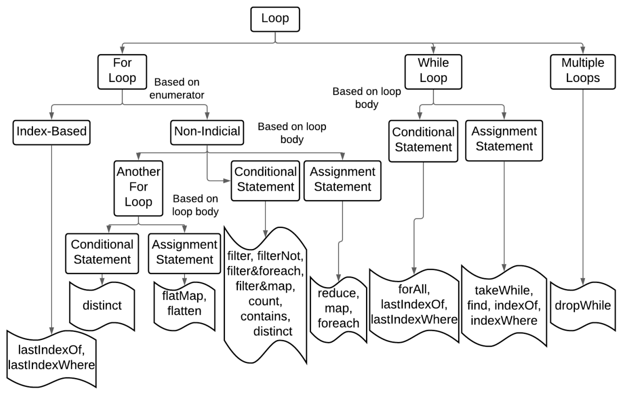

Figure 3. Categorization of loops and possible refactorings.

```
if (a == 0) f(a)
else if (b==10 && a==
    5)
  f(a+b)
else a + b + c
```

```
(a, b) match {
case (0,_) => f(a)
case (5,10) =>
   f(a + b)
case (_,_) =>
   a + b + c
}
```

Figure 4. Conditional: Original and refactored code.

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

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

Figure 5. Loop+Conditional: Original and refactored code.

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.

The application 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. Table 1 shows few examples of the possible refactorings. 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.

ScalaRT also identifies and refactors if-else statements and nested else-if statements into pattern match expressions. The source code syntax tree is traversed and when a conditional expression is detected, a 'map' of predicates and corresponding main clauses is generated, which is then analysed in two ways. First, the predicates (in case of nested expressions), are analysed together, to come up with the match expression, which can be a single variable or a tuple. In Figure 4, it can be seen that the match expression *(a,b)* is a 2-tuple. In the second analysis, the predicates are individually assessed and transformed into the signature 'case clauses' of a typical pattern-match body. The main clauses stored with their antecedents in the map earlier, are now plugged in to the corresponding case clauses. Thus the entire syntax tree-body of the pattern match expression is constructed.

Almost any kind of nested else-if conditional statement can be refactored 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 match will likely defeat the purpose of this refactoring, which is to improve 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 a conditional expression enclosed within a loop, refactorable to a combination of sequence method 'map' and pattern match expression is also covered in conditional refactorings, as shown in Figure 5.

ScalaRT currently stands at 1300+ lines of meta-program. Our future goals include adding more refactorings, and evaluating the original and refactored program with respect to a code-evaluation metric, thus coming up with a code-quality check system. Furthermore, our analyses and transformations are general enough and extensible to other languages that support functional patterns as well.

Tool Output	Loop code snippet

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

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

Table 1. Refactored code output and original code snippet.

References

[1] Michael Bevilacqua-Linn. 2013. *Functional Programming Patterns.* The Pragmatic Programmers, LLC.

[2] ScalaMeta.Org. 2016. Scalameta- Library to read, analyze, transform and generate Scala programs. (2016). Retrieved June 2021 from https://scalameta.org/

[3] Jyothi Vedurada and V Krishna Nandivada. 2017. Refactoring Opportunities for Replacing Type Code with State and Subclass. In *Proceedings of the 39th International Conference on Software Engineering Companion (ICSE-C '17).* IEEE Press, 305–307. DOI: http://dx.doi.org/10.1109/ICSE-C.2017.97