Mining the Use of Higher-Order Functions: An Exploratory Study on Scala Programs

Yisen Xu[1], Fan Wu[2], Xiangyang Jia[1], Lingbo Li[2], and Jifeng Xuan[1⋆]

[1] Wuhan University, China {xuyisen, jxy, jxuan}@whu.edu.cn
[2] Turing Intelligence Technology Limited, UK {fan, lingbo}@turintech.ai

Abstract. A higher-order function takes one or more functions as inputs or outputs to support the generality of function definitions. In modern programming languages, higher-order functions are designed as a feature to enhance the usability and scalability. Abstracting higher-order functions from existing functions decreases the number of similar functions and improves the code reuse. However, due to the complexity, defining and calling higher-order functions are not widely used in practice. In this paper, we investigate the use of higher-order functions in Scala programs. We collected 20 Scala projects from GitHub with the most stars and conducted an exploratory study via answering five research questions of using higher-order functions, including the data scale, the definitions and calls, the overlap with lambda expressions, the developer contribution, and the factor that affects the function calls. Our study mainly shows five empirical results about the common use of higher-order functions in Scala programs: higher-order functions are not widely-used in Scala programs; most of higher-order functions are defined and called by the same developers; there exists a small overlap between calling higher-order functions and calling lambda expressions; similar to the practice in many fields, top 20% of developers have contributed 76% of definitions and 78% of calls of higher-order functions; and the number of calls of higher-order functions correlates most with the number of executable lines of code and the code complexity. This study can be viewed as a preliminary result to understand the use of higher-order functions and to trigger further investigation in Scala programs.

Keywords: Scala programs · Higher-order functions · GitHub · Exploratory study · Correlation analysis · Code reuse

1 Introduction

A higher-order function is a function that takes one or more functions as parameters or returns a function as a result. The concept of higher-order functions is derived from mathematics and can be intuitively considered as a *function of functions*. Using a higher-order function can increase the generality of source code and reduce the redundancy by discarding functions that share the same

⋆ Corresponding author: Jifeng Xuan

functionality without using the same types of parameters. Abstracting higher-order functions from existing functions can be further leveraged to support automated code reuse and code generation via reduce the search space of potential functions.

Many program languages support programming with higher-order functions, e.g., C++ 11, Java 1.8, Python, and Scala. Using higher-order functions eases the design and the implementation of programs. However, defining and calling a higher-order function is complex since functions are introduced as parameters or returned results. Such complexity hurts the possibility of widely using higher-order functions.

Developers employ higher-order functions to enhance the design of source code in following ways. First, common patterns are encapsulated into higher-order functions to improve reusability in source code. The design of higher-order functions is the abstraction of common programming patterns by taking functions as the input or the output [27]. Second, higher-order functions can be leveraged to implement polymorphism in source code. Polymorphism is the ability of the same behavior to have many different manifestations or forms [6]. For instance, the mapping function of a collection in Scala programs, such as `Array.map()`, is a higher-order function, which supports polymorphism by receiving different functions as parameters.[3] Third, for complex calculation like high-precision control systems, developers can use higher-order functions to keep the source code of calculation simple and clear. For example, the tensor data with any non-hierarchical storage format and arbitrary number of dimensions can be handled by the recursive multi-index algorithms implemented in higher-order functions [4].

The direct support of higher-order functions is one of the important features of the Scala language. As an emerging object-oriented programming language, Scala was initially proposed to address the design drawbacks of the Java language. The design of Scala contains the implementation of generics, the forced use of object-oriented programming, and the implementation of limited support for component abstraction and composition [11]. To attract developers using existing programing languages like Java and C#, Scala has been designed to provide the support of type systems and the compatibility of functional programming and object-oriented programming [23]. Such design makes the Scala language be widely used in many development scenarios, e.g., the rapid web development and the construction of distributed systems. For instance, Nystrom [22] has presented a Scala framework for experimenting with super-compilation techniques; Kroll et al. [13] have proposed the Scala platform that conducts a straightforward and simplified translation from a formal specification to source code.

How do developers use higher-order functions in Scala programs? In this paper, we conducted an exploratory study on the use of higher-order functions in 20 Scala projects with the most stars in GitHub. We leverage the static analysis

[3] `Array.map()` in Scala, `http://www.scala-lang.org/api/2.12.8/scala/Array.html#map[B](f:A=>B):Array[B]`.

to explore the definitions and calls of higher-order functions by developers and answered five research questions,

- **RQ1. How many higher-order functions exist in Scala projects?** We give a basic statistics on the higher-order functions in 20 Scala projects and show the existence of higher-order functions. The empirical result shows that higher-order functions are not widely-used in Scala programs. Meanwhile, most of higher-order functions are defined in a simple way, but there indeed exist complex definitions.
- **RQ2. How are higher-order functions defined and called?** We briefly categorize the definitions of higher-order functions according to inputing or outputting functions. Most of definitions of higher-order functions only takes functions as input without functions as output; most of higher-order functions are called by the same developers who have defined the functions.
- **RQ3. How much is the overlap of calling higher-order functions and calling lambda expressions?** We count the number of calling higher-order functions and calling lambda expressions to reveal their difference in practice. The result shows that there exists a small overlap between calling higher-order functions and calling lambda expressions; that is developers treat higher-order functions and lambda expressions in a different way.
- **RQ4. How do developers contribute to write higher-order functions?** We analyze the number of higher-order functions defined and called by each developer. Similar to the practice in many fields, we find out that top 20% of developers have contributed 76.15% function definitions and 78.06% calls of higher-order functions.
- **RQ5. Which factor affects the calls of higher-order functions?** We further analyze the factors that affect the calling of higher-order functions. We mainly examine three measures, i.e., the LoC, the complexity, and the warnings in the code style. The Spearman's rank correlation coefficient indicates that the number of calls of higher-order functions correlates with the number of executable lines of code and the Cyclomatic complexity.

This paper makes the following major contributions:

1. We mined 20 Scala projects with the most stars in GitHub and collected 5494 higher-order functions which are contributed by 396 developers.

2. We designed an exploratory study on using higher-order functions via answering five research questions, including the data scale, the definitions and calls, the overlap with lambda expressions, the developer contribution, and the factor that affects the function calls.

3. We empirically investigate the use of higher-order functions in Scala programs. We found that developers tend to define simple higher-order functions; the LoC and the code complexity correlates most with the number of calls of higher-order functions.

The rest of this paper is organized as follows. Section 2 shows the background and motivation of studying higher-order functions. Section 3 presents the study setup, including five research questions and the data preparation. Section 4 de-

scribes the results of our exploratory study. Section 5 discusses the threats to the validity. Section 6 lists the related work and Section 7 concludes.

2 Background and Motivation

Scala is a multi-paradigm programming language that provides the support of functional programming and a strong static type system [23]. Scala, like Java, is object-oriented and the source code of Scala is designed to be compiled into Java bytecode. This enables the compiled Scala code to be directly executed on the Java virtual machine. Besides the object-oriented features of Java, Scala shares many features of functional programming languages with Standard ML and Haskell, including currying, type inference, and immutability. A higher-order function, a typical feature of functional languages, makes the Scala code succinct. Using higher-order functions in Scala can improve the abstraction and simplification of function design, which is not originally supported in Java.[4]

A function plays an important role in Scala. Calling a function is similar to using a variable or an object. A higher-order function takes functions as input or returns a function as output. The input or output function can be any function, including higher-order functions. If a function serves as input, it abstracts the common patterns of potential parameters of the higher-order function; if a function serves as output, it increases the diversity of the returned results. Meanwhile, a higher-order function can have both functions as parameters and functions as returned results. The flexible use of functions in Scala can hurt the readability [29]. For instance, if a higher-order function returns a function, the type of the returned function can be omitted from the definition of higher-order function.

Fig. 1 shows an excerpt of a real-world higher-order function `readModifiers()` in Project `lampepfl/dotty`.[5] The higher-order function `readModifiers()`, locating in `dotty.tools.dotc.core.tasty.TreeUnpickler`, is designed to deserialize several modifiers of an Abstract Syntax Tree (AST) into a triplet, which contains of a set of flags, a list of annotations, and a boundary symbol. A *modifier* of an AST is a qualifier for the access, such as `private` or `protected` for a variable; a *boundary symbol* is the scope of the accessible variable, e.g., a package name or a class name of the variable. A *flag* is a value of a long integer that reflects a particular modifier; then the variable `flags` at Line 8 indicates all the modifiers of an AST.

The definition of this higher-order function contains five input parameters and a return type. Five parameters are `end`, `readAnnot`, `readWithin`, `defaultWithin`, and `ctx`, respectively. Among these five parameters, two parameters `end` at Line 2 and `defaultWithin` at Line 4 are objects of Class `Addr` and Class `WithinType`, respectively; another two parameters `readAnnot` at Line 3 and `readWithin` at Line 4 are function parameters: the function of `readAnnot: Context => Symbol => AnnotType` and the function of `readWithin: Context => WithinType`;

[4] Java supports higher-order functions since its Version 8.0 in 2014.
[5] Project dotty, `http://github.com/lampepfl/dotty`.

```
1  def readModifiers[WithinType, AnnotType]
2      (end: Addr,                                              // Parameter 1
3       readAnnot: Context => Symbol => AnnotType,              // Parameter 2: function
4       readWithin: Context => WithinType,                      // Parameter 3: function
5       defaultWithin: WithinType)                              // Parameter 4
6      (implicit ctx: Context):                                 // Parameter 5
7  (FlagSet, List[Symbol=>AnnotType], WithinType)={ // Return type
8      var flags: FlagSet = EmptyFlags
9      var annotFns: List[Symbol => AnnotType] = Nil
10     var privateWithin = defaultWithin
11     while (currentAddr.index != end.index) {
12         def addFlag(flag: FlagSet) = {
13           flags |= flag
14           readByte()
15         }
16         nextByte match {
17           case PRIVATE => addFlag(Private)
18           ...
19           case PRIVATEqualified =>
20             readByte()
21             privateWithin = readWithin(ctx)
22           case PROTECTEDqualified =>
23             addFlag(Protected)
24             privateWithin = readWithin(ctx)
25           case ANNOTATION =>
26             annotFns = readAnnot(ctx) :: annotFns
27           case tag =>
28             assert(false, s"illegal␣modifier␣tag␣$tag␣at␣$currentAddr,␣end␣=␣$end")
29         }
30     }
31     (flags, annotFns.reverse, privateWithin)                 // Return statement
32 }
```

Fig. 1. Excerpt of a real-world higher-order function `readModifiers()` from Class `dotty.tools.dotc.core.tasty.TreeUnpickler` in Project `lampepfl/dotty`. The definition of this function is to read a modifier list into a triplet of flags, annotations, and a boundary symbol.

the last parameter `ctx` at Line 6 belongs to the function currying that transforms a function with multiple parameters into a sequence of functions, where the `implicit` keyword indicates that the parameter is optional.[6] As shown in Fig. 1, the second parameter `readAnnot` at Line 3 of the function `readModifiers()` is a higher-order function, which receives a `Context` object as input and returns an anonymous function `Symbol => AnnotType` and the function `Symbol => AnnotType` takes a `Symbol` object as a parameter and returns an `AnnotType`

[6] Scala currying, `http://docs.scala-lang.org/tour/currying.html`.

object. This parameter is called at Line 26. The third parameter `readWithin` at Line 4 is a first-order function, which receives a `Context` object as input and returns a `WithinType` object. This parameter is called at Line 21 and Line 24.

The return type of the function `readModifiers()` at Line 7 is a triplet of flags, annotations and a boundary symbol. In the return type, the second return value is a list of functions; that is, `List[Symbol => AnnotType]` denotes a list of functions, each of which takes a `Symbol` object as input and returns an `AnnotType` object as output. The return statement of the function `readModifiers()` locates at Line 31, which returns an instance of the above return type at Line 7.

Motivation. Higher-order functions are widely used in the development of many applications [35,5,22]. However, there is no prior study that investigates the use of higher-order functions. For instance, in a mature project, how many functions belong to higher-order functions? How many times is a higher-order function called? Who has defined or called the higher-order function? Which factor impacts the number of calls? In this paper, we extracted data of 20 popular Scala projects from GitHub and conducted an exploratory study on understanding the use of higher-order functions in Scala programs.

3 Study Setup

In this section, we first present the data preparation of our study and then describe the design of five research questions.

3.1 Data Preparation

Our study aims to understand the use of higher-order functions in Scala programs. We mined 20 Scala projects and extracted data for further analysis, including function definitions, calls, developers who have written the functions, and the function complexity.[7] We employed the static analysis tool SemanticDB to extract semantic structures, such as types and function signatures. SemanticDB is a library suite for program analysis of Scala source code.[8] The main steps of data preparation are listed as follows.

Project selection. We sorted all Scala projects in GitHub according to the stars.[9] A project with many stars indicates that the project is identified by developers because of the usage and the quality. Then we selected top projects with the most stars. Applying SemanticDB to a project requires the compatible configuration of the project. Thus, we skipped the projects that cannot be parsed by SemanticDB and selected the other top 20 projects for the experiment. In detail, seven projects were skipped: since SemanticDB can only support the static analysis of Scala 2.11 or Scala 2.12, five projects that implemented in Scala 2.10 were

[7] The collected data in this study are publicly available, `http://cstar.whu.edu.cn/p/scalahof/`.

[8] SemanticDB, `http://scalameta.org/docs/semanticdb/guide.html`.

[9] Scala projects with stars, `http://github.com/search?l=Scala&o=desc&q=scala&s=stars&type=Repositories`.

Table 1. Summary of 20 Scala projects in GitHub in the study. For the sake of space, each project will be denoted by its abbreviation in following sections.

Project	Abbr.	#Star	LoC	Project description
scala/scala	scala	11.4k	143.6k	The Scala programming language
playframework/playframework	framework	11.0k	41.5k	A web framework for building scalable applications with Java and Scala
gitbucket/gitbucket	gitbucket	7.6k	19.1k	A Git platform powered by Scala
twitter/finagle	finagle	7.0k	63.0k	An extensible RPC system for the Java JVM
yahoo/kafka-manager	kafka	6.9k	10.5k	A tool for managing Apache KafKa
ornicar/lila	lila	5.2k	70.3k	A free server for the online chess game
rtyley/bfg-repo-cleaner	bfg	5.0k	1.5k	A simple and fast tool for cleansing bad data out of Git repository
gatling/gatling	gatling	4.2k	25.6k	A highly capable load testing tool
scalaz/scalaz	scalaz	4.1k	35.4k	A Scala library for functional programming
apache/incubator-openwhisk	openwhisk	3.9k	18.7k	A cloud-first distributed event-based programming service
sbt/sbt	sbt	3.8k	34.9k	A build tool for Scala, Java, and other languages
lampepfl/dotty	dotty	3.3k	387.1k	A Scala compiler
twitter/scalding	scalding	3.1k	29.6k	A Scala API for a Java tool named cascading
milessabin/shapeless	shapeless	2.7k	30.5k	A Scala library for generic programming
scalatra/scalatra	scalatra	2.4k	8.4k	A tiny, Sinatra-like web framework
spark-jobserver/spark-jobserver	jobserver	2.2k	7.6k	A REST job server for Apache Spark
twitter/util	util	2.2k	27.4k	A collection of core JVM libraries of Twitter
slick/slick	slick	2.2k	20.8k	A Scala library for database querying and accessing
lagom/lagom	lagom	2.1k	21.8k	A framework for building reactive microservice systems in Java or Scala
lihaoyi/Ammonite	ammonite	1.9k	8.6k	A Scala tool for scripting purposes
Total		92.2k	1005.9k	

skipped, including Projects `scala-js/scala-js`, `scala-native/scala-native`, `scalanlp/breeze`, `spray/spray`, and `pocorall/scaloid`; another one project `intel-analytics/BigDL` that is mixedly implemented with Java and Scala is skipped due to the failure of configuring with SemanticDB; we also removed Project `fpinscala/fpinscala` that is a supplement material of practices in a book since the project is not a real software project. Table 1 lists the summary of 20 Scala projects in the study.

Function extraction. We collected definitions and calls of functions via SemanticDB. We filtered out the files that are not written in Scala. For each project, we used SemanticDB to extract all the function definitions and identified whether there exists a function in the input or the output. That is, we collected all the definitions of higher-order functions in each project. SemanticDB can generate a semantic database that contains all classes, functions, objects, and variables as well as their positions in source code. Then we collected all function calls of the higher-order functions by parsing the above semantic database. As shown in Table 1, in all the 20 projects, 11,671 Scala files and 1,100 KLoC are examined in our study; 5494 definitions of higher-order functions with 21,782 calls are recorded for further analysis.

Developer extraction. We extract developers who have written the function definitions and calls to understand the use of higher-order functions. We used the Git API to extract the Git log and collected historical commits that

relate to the changes of function definitions and calls. For each of such commits, we extracted the developer (including the name and the e-mail) who submitted it, the timestamp, the added changes and positions. For a definition of a higher-order function, we sorted all related changes according to the timestamps and identified the developer who wrote the first change as its original author [37]; for a function call, we similarly identified the first developer as its original author among all changes that added or modified the function call.

LoC measurement. We leverage the LoC to directly measure the lines of code of a higher-order function. The *LoC* is the number of executable lines of code without blank or comments; to count LoC, we measured the lines of Scala code inside a higher-order function. We used LoC as a simple way to briefly record the length of code.

Code complexity measurement. We use the Cyclomatic complexity to measure the complexity of the definition of a higher-order function. The *Cyclomatic complexity* is a software metric of linearly independent paths [18]; to count the Cyclomatic complexity, we used the static analysis tool Scalameta to parse Scala files and construct the AST.[10] Then we identified the Cyclomatic complexity of each higher-order function by traversing its AST.

Code style measurement. We leverage the number of suspected issues of code style of Scala functions to measure the code quality of a higher-order function. We define *#StyleWarnings* as the number of suspected issues of code style via an off-the-shelf checking tool of the code style, ScalaStyle.[11] For each definition of a higher-order function, we count the number of issues inside the definition.

3.2 Research Questions

Our work is to understand the use of higher-order functions in Scala programs. We designed RQs to analyze the definitions and callings in five categories: the data scale, the definitions and calls, the developer contribution, and the factor that affects the function calls.

RQ1. How many higher-order functions exist in Scala projects? Higher-order functions are introduced to many programming languages. However, the ratio of higher-order functions among all functions is unclear. We designed RQ1 to reveal the data scale of using higher-order functions, i.e., how many higher-order functions are there in Scala programs. Intuitively, a complex higher-order function could be seldom called. We analyze the definitions and calls of higher-order functions in RQ1.

RQ2. How are higher-order functions defined and called? In general, a higher-order function can input a function as a parameter and/or output a function as a returned result. This leads to three kinds of higher-order functions based on the input and the output. Are higher-order functions only called by the authors who have written the function definitions? We aim to investigate how

[10] Scalameta, `http://scalameta.org/`.
[11] ScalaStyle, `http://www.scalastyle.org/`.

many functions are called by the same developers or by the other developers in RQ2.

RQ3. How much is the overlap of calling higher-order functions and calling lambda expressions? Higher-order functions and lambda expressions are two concepts of supporting code reuse in Scala and other modern languages. The lambda expressions can be used as input parameters of higher-order functions. Developers feel confusing when using higher-order functions and lambda expressions [30], [31]. Thus, we analyze the overlap of calling higher-order functions and lambda expressions in RQ3.

RQ4. How do developers contribute to write higher-order functions? Besides source code, developers play an important role of using higher-order functions. In RQ3, we aim to examine how many developers are involved in using higher-order functions.

RQ5. Which factor affects the calls of higher-order functions? Higher-order functions are expected to abstract the use pattern of functions [11]. In RQ4, we investigate the potential factors that affects the number of calls of higher-order functions. Empirical results of RQ4 can provide a way to understand how to increase the number of function calls.

4 Empirical Results

We conducted experiments on higher-order functions and investigated five research questions. The results and findings based on these research questions are listed as follows.

4.1 RQ1. How many higher-order functions exist in Scala projects?

Goal. We aim at answering RQ1 and exploring the data scale of using higher-order functions. The LoC, the Cyclomatic complexity, and the warnings in the code style are leveraged to directly measure the source code of Scala functions.

We collected definitions and calls of all higher-order functions in 20 Scala projects. Table 2 lists the ratios of defining and calling higher-order functions among all functions. In total, there exist 5494 definitions and 232727 calls of these higher-order functions. Among 20 projects, higher-order functions account for 6.33% of function definitions in average and 7.49% of function calls. The percentage of higher-order functions among 20 projects is unstable: the ratio of definitions of higher-order functions ranges from 1.20% to 22.64% while the ratio of calls ranges from 0.54% to 28.81%. In eight projects, over 5% of functions are defined as higher-order functions; in seven projects, over 10% of function calls are for higher-order functions. Among 20 projects, Project `scalaz` reaches the highest ratio of defining and calling higher-order functions; Project `kafka` reaches the lowest ratio of definitions while Project `openwhisk` reaches the lowest ratio of calls.

As shown in Table 2, only 5 out of 20 projects have a higher ratio than the average ratio 6.36% when we consider the ratio of the definitions of higher-order

Table 2. Numbers of definitions and calls of higher-order functions. Column "Ratio" denotes the ratio of the number of higher-order functions dividing the number of all functions.

Abbr.	Function definitions			Function calls		
	#All	#Higher-order	Ratio (%)	#All	#Higher-order	Ratio (%)
scala	24779	1216	4.91%	96002	5162	5.38%
framework	3095	196	6.33%	5474	266	4.86%
gitbucket	1056	61	5.78%	3218	543	16.87%
finagle	6178	157	2.54%	5904	204	3.46%
kafka	753	9	1.20%	1216	25	2.06%
lila	5733	165	2.88%	11240	505	4.49%
bfg	119	10	8.40%	144	5	3.47%
gatling	2718	118	4.34%	4186	345	8.24%
scalaz	8184	1853	22.64%	14594	4191	28.72%
openwhisk	1409	38	2.70%	7536	41	0.54%
sbt	4076	436	10.70%	13638	1989	14.58%
dotty	10372	256	2.47%	41312	770	1.86%
scalding	4174	292	7.00%	6285	846	13.46%
shapeless	1936	50	2.58%	2416	46	1.90%
scalatra	1480	57	3.85%	1823	33	1.81%
jobserver	620	13	2.10%	698	30	4.30%
util	2777	176	6.34%	3278	500	15.25%
slick	2881	161	5.59%	4075	391	9.60%
lagom	1813	70	3.86%	1397	35	2.51%
ammonite	792	72	9.09%	1211	133	10.98%
Total	84945	5406	6.36%	225647	16060	7.12%

functions among all functions; only 8 out of 20 projects have a higher ratio than the average ratio 7.12% when we consider the ratio of the calls of higher-order functions among all functions. The ratio of defining and calling higher-order functions indicates that higher-order functions account for a lower proportion among all functions.

1) Executable Lines of Code

We used the LoC, the Cyclomatic complexity, and the warnings in the code style to measure the complexity of definitions of higher-order functions. Fig. 2 presents the accumulative percentage of the definitions of higher-order functions by counting LoC of no less than 10 lines. As shown in the figure, 17 out of 20 projects have over 10% of definitions of higher-order functions with over 10 lines; One project, `openwhisk` has over 50% of definitions of higher-order functions with over 10 lines. Considering the number of lines, 5 out of 20 projects have over 10% of definitions of higher-order functions with over 20 lines; 3 projects have over 10% with over 30 lines. In 7 out of 20 projects, i.e., `framework`, `scalaz`, `util`, `slick`, `bfg`, `shapeless`, and `scalatra`, there exist no definitions of higher-order functions with over 30 lines.

To further understand the distribution of LoC of higher-order functions, we illustrated the violin-plots to present the probability density of LoC in each project in Fig. 3. In each violin-plot, the LoC value with the broadest line indicates the LoC that appears for the most times. Among 20 projects, 17 projects

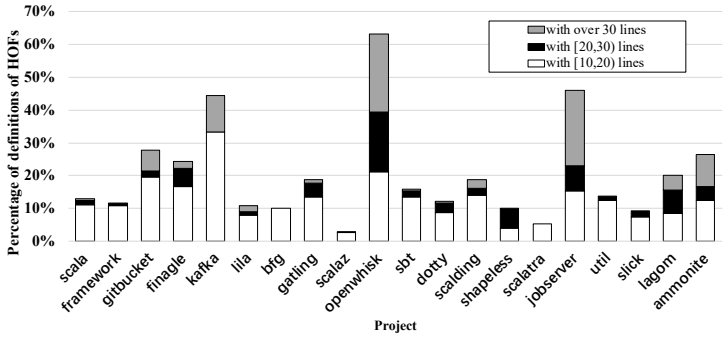

Fig. 2. Accumulative percentage of function definitions for higher-order functions by counting LoC of no less than 10 lines. Term *HOFs* is short for higher-order functions.

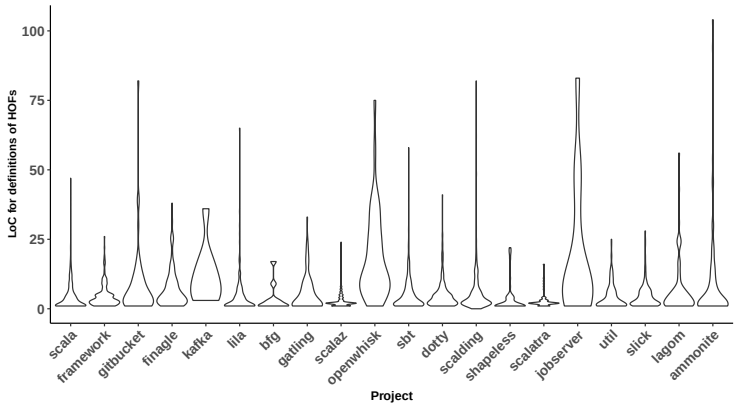

Fig. 3. Violin-plots of LoC for function definitions of higher-order functions. The width of each bar is equal, which denotes the maximum number of definitions with the same LoC inside one project.

show a similar shape, where the data are mainly concentrated at the bottom; the other 3 projects, i.e., `kafka`, `openwhisk`, and `jobserver`, have not concentrated higher than the bottom. The highest LoC reaches 104 in Project `ammonite`. We concluded that the LoC of higher-order functions in most of the projects is distributed at the bottom, i.e., the LoC less than 10.

2) Cyclomatic Complexity

Besides the lines of executable code, we leveraged the Cyclomatic complexity to quantify the complexity of function definitions by counting the number of linearly independent path. Fig. 4 presents the accumulative percentage of definitions of higher-order functions in Cyclomatic complexity. Among 20 projects, 19 projects except `lagom`, have over 10% of definitions of higher-order functions with the complexity of two or more; 11 projects and 6 projects have over 20%

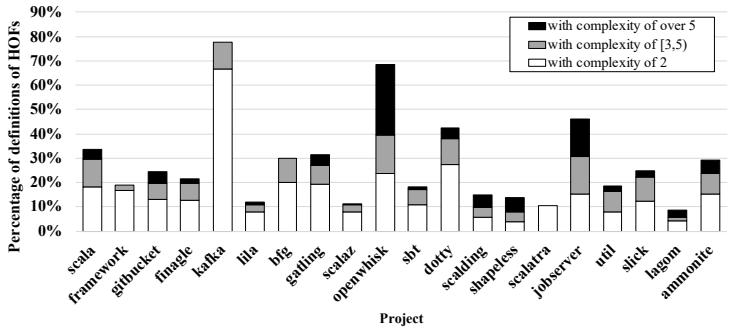

Fig. 4. Accumulative percentage of function definitions for higher-order functions by counting Cyclomatic complexity of more than one.

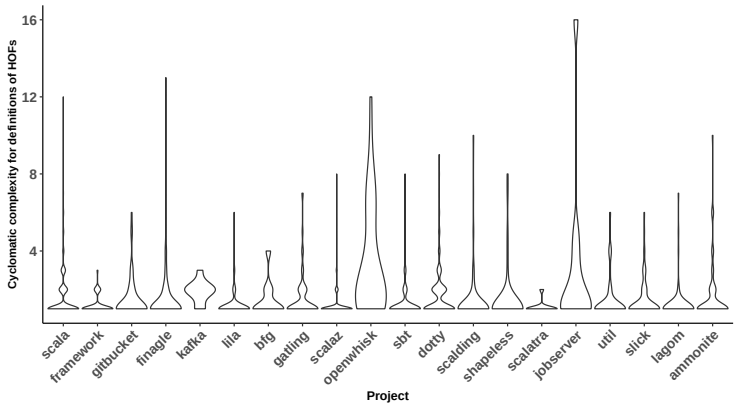

Fig. 5. Violin-plots of Cyclomatic complexity for the definitions of higher-order functions. The width of each bar is equal, which denotes the maximum number of definitions with the same complexity inside one project.

and 30% of definitions of higher-order functions of two or more, respectively. As shown in Fig. 4, 6 out of 20 projects have over 10% of definitions of higher-order functions with the complexity of over three while two projects have over 10% of definitions of higher-order functions with the complexity of over five.

Fig. 5 presents the violin-plots of Cyclomatic complexity for the definitions of higher-order functions. From the figure, 17 projects have a similar distribution, where the data are mainly concentrated at the bottom. Most of higher-order functions tend to be simple code structure with the Cyclomatic complexity of less than five. Three projects behave different: plots of Projects `kafka`, `openwhisk`, and `jobserver` aggregate in the middle, not the bottom. The highest complexity reaches 16 in Project `jobserver`.

Through the analysis of the LoC and complexity of higher-order functions, we find that most higher-order functions favor low LoC and complexity. We note

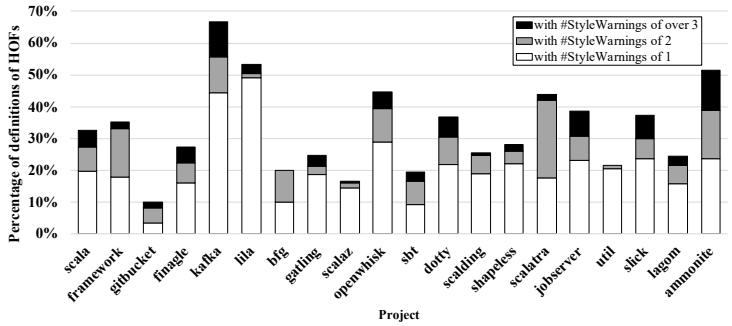

Fig. 6. Accumulative percentage of function definitions for higher-order functions by counting the warnings in the code style over zero.

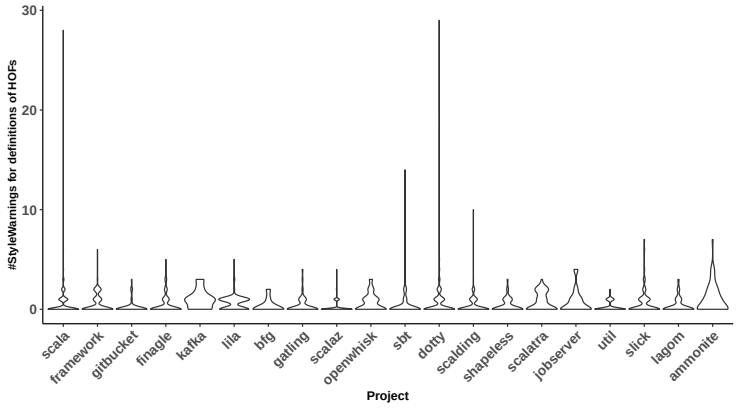

Fig. 7. Violin-plots of warnings in the code style for the definitions of higher-order functions. The width of each bar is equal, which denotes the maximum number of definitions with the same number of warnings in the code style inside one project.

that a higher-order function can actually represent a first-order function, which contains several times of lines of code, compared with the higher-order function [14]; the linear increment of the LoC or complexity in a higher-order function may represent an exponential increment of the LoC or complexity in a first-order function.

3) Warnings of the Code Style

The code style is also used to measure the source code. Bacchelli and Bird [3] and Georgios et al. [9] have shown that code style issues highly affect the code review and code integration. Zou et al. [38] found that the inconsistency of the code style can delay the process of merging new changes. In this study, we used the number of reported warnings of the code style to measure the source code of higher-order functions.

Fig. 6 presents the accumulative percentage of definitions of higher-order functions by counting the warnings in the code style. Among 20 projects, 19 projects except `gitbucket` have over 10% of definitions of higher-order functions with the warnings in the code style of one or more; 16 projects and 10 projects have over 20% and 30% of definitions of higher-order functions with the warnings in the code style of one or more, respectively. As shown in the figure, 11 out of 20 projects have over 10% of definitions of higher-order functions with the warnings of two or more; two projects, `kafka` and `ammonite`, have over 10% of definitions of higher-order functions with the code style of over three. One project, `kafka`, have over 65% of definitions of higher-order functions with the code style of one or more. This observation reveals that most of higher-order functions in Project `kafka` have code style issues.

Fig. 7 presents the violin-plots of warnings in the code style for the definitions of higher-order functions. Among 20 projects, 14 projects have a similar distribution, where the data are mainly concentrated at the bottom. That observation of most of higher-order functions with the warnings of zero shows that these higher-order functions have no code style issues. Six projects behave different: plots of Projects `kafka`, `lila`, `openwhisk`, `scalatra`, `jobserver`, and `ammonite` aggregate in the middle. The highest number of warnings in the code style reaches 29 in Project `dotty`; that is a higher-order function contains 29 reported warnings of the code style.

The warnings of in the code style exist in more than 10% of higher-order functions among all projects. However, we should note that the not all code style issues relates to the quality; this may result in the ignorance of code style issues in daily development [19].

> Among 20 Scala projects in our study, 6.33% of functions are defined as higher-order functions in average. Most of function definitions tend to be simple while there indeed exist complex definitions: 17 out of 20 projects have over 10% of definitions of higher-order functions with over 10 lines; 19 projects have over 10% of definitions with the complexity of two or more. Code style issues widely exist: 19 projects have over 10% of definitions with the code style of at least one warning. Developers or project managers may need to promote the acceptance of using higher-order functions in Scala projects in future.

4.2 RQ2. How are higher-order functions defined and called?

Goal. We aim to understand how developers define and call higher-order functions. The number of function calls can be used to indicate the popularity of using higher-order functions.

We divide all definitions of higher-order functions into three types by checking whether the input or the output contains a function,

- `Type I`, a function definition takes at least one function as a parameter without returning functions;

```
/* Type I */
def closed[R](fn:()=>R)={
  val closure = Local.save()
  val save = Local.save()
  Local.restore(closure)
  try fn()
  finally Local.restore(save)
}
```

```
/* Type II */
def closed[R]:()=>R ={
  val closure = Local.save()
  () => R
}
```

```
/* Type III */
def closed[R](fn:()=>R):() => R = {
  val closure = Local.save()
  () =>
    {
      val save = Local.save()
      Local.restore(closure)
      try fn()
      finally Local.restore(save)
    }
}
```

Fig. 8. Examples of higher-order functions in Type I, Type II, and Type III.

- `Type II`, a function definition returns at least one function without inputing a function;
- `Type III`, a function definition takes at least one function as a parameter and returns functions.

Fig. 8 shows examples of three types of definitions of higher-order functions. The example in Type I takes a function `fn()` as input and has no specific output; the example in Type II has no input and returns an anonymous function as output; the example in Type III takes the function `fn()` as input and returns an anonymous function as output.

We briefly present the distribution of definitions and calls of three defined types of higher-order functions. Fig. 9 presents the percentage of function definitions of higher-order functions in all projects by categorizing the definition types. Among 20 projects under consideration, we can observe that 14 projects contain over 80% of definitions in `Type I` while 7 projects contain 90% of definitions in `Type I`. This observation shows that `Type I` of higher-order functions are more frequently defined than the other two types. In Project `kafka`, the percent of `Type I` of higher-order functions is highest and reaches 100%; in Project `bfg`, the percent is lowest and is 50%. Meanwhile, in 12 projects, over 10% of definitions of higher-order functions belong to `Type II` and in 5 projects, over 20% of definitions of higher-order functions belong to `Type II`. In addition, `Type III` of higher-order functions account for over 10% in 2 projects.

We further show the percentage of calls of higher-order functions in three types in Fig. 10. We can find that 14 out of 20 projects have over 80% of calls of higher-order functions in `Type I` while 11 projects contain 90% of definitions in `Type I`. In one project, `kafka`, the percentage of `Type I` of higher-order functions is the highest and reaches 100%. Meanwhile, in 9 projects, over 10% of

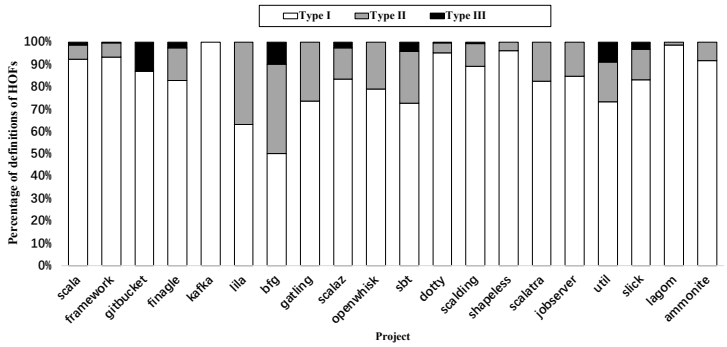

Fig. 9. Percentage of function definitions of higher-order functions in all projects by categorizing the definition types.

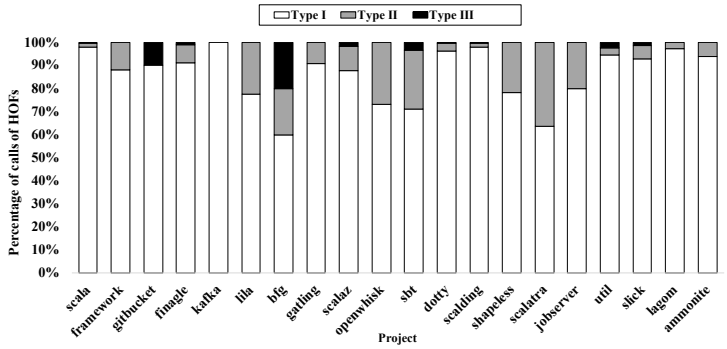

Fig. 10. Percentage of function calls of higher-order functions in all projects by categorizing the definition types.

calls of higher-order functions belong to `Type II`, and in 7 projects, over 20% of calls of higher-order functions belong to `Type II`. Only one project, `bfg`, has over 10% of calls of higher-order functions in `Type III`. According to Fig. 9 and Fig. 10, `Type I` accounts for the highest percent among all the definitions and calls of higher-order functions.

Among the large number of function calls, *how many functions are only called by the same developers who design the functions*? We further investigate this question to understand the overlap of developers between function definitions and calls. As mentioned in Section 3.1, we collected developer information via the Git API. For the sake of simplification, we identified the first developer, who has created the definition or the call of higher-order functions, as the author of defining or calling higher-order functions.

Fig. 11 illustrates the percentage of definitions in three categories: a definition that is only called by the author of the definition, a definition that is only called

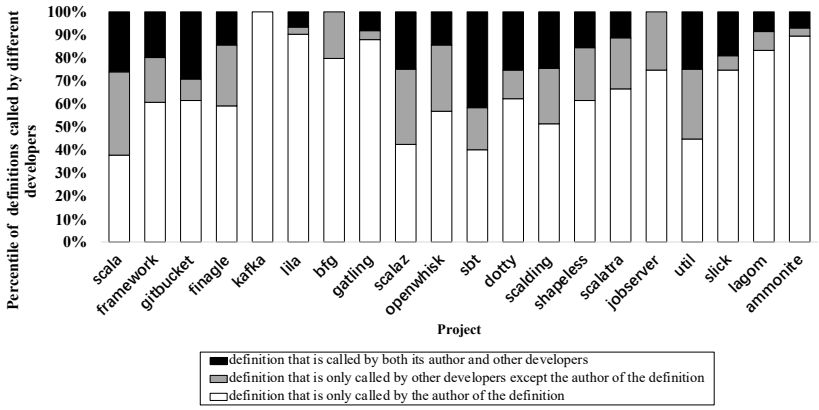

Fig. 11. Percentage of function definitions that are called by different types of developers.

by other developers except the author of the definition, and a definition that is called by both its author and other developers. Among 20 projects under consideration, we can observe that 16 projects have over 50% of definitions of higher-order functions which are only called by the author of these definitions while 6 projects contain 80% of definitions in the first category. In Project `kafka`, the percent of the first category of higher-order functions is highest and reaches 100%. This observation shows that the definitions of higher-order functions are more frequently called by the author of these definition than other developers. In addition, in 11 projects, over 20% of definitions of higher-order functions belong to the second category and in 8 projects, over 20% of definitions of higher-order functions belong to the third category.

Besides the function definitions, we refined the function calls of each project according to the overlap of developers of functions definitions and calls. Table 3 lists the number of function calls based on three types of definitions and the the overlap of developers. Among 60 mini-bar charts, there exist 11 blank ones, including 2 blank mini-bar charts in `Type II` and 9 in `Type III`. Among the 49 non-blank mini-bar charts, 33 charts show that most of calls are made by the authors of functions definitions; 1 chart shows that most of calls are made by other developers; and 15 charts shows that most of calls are made by both the authors of definitions and other developers. This fact indicates that most of calls of higher-order functions are made by the same developers who have written the higher-order functions. That is, there exist a large percent of higher-order functions that are not designed for collaborative development among developers.

As shown in Table 3, there also exist 3237 function calls of higher-order functions that are made by developers other than the authors of definitions, including 3009 calls of `Type I`, 169 calls of `Type II`, and 59 calls of `Type III`. This observation indicates that developers have maintained the collaboration between defining and using functions. We can also observe that among all the

Table 3. Function calls of higher-order functions based on the types of `Type I`, `Type II`, and `Type III` and the overlap of the authors of definitions and calls. The number of calls of each definition type is listed, including the number of functions that are called only by the authors of the definition, only by developers other than the authors, and both. We illustrate mini-bar charts to briefly compare the number of function calls inside each definition type of each project. Sub-columns "All", "Self", "Others", and "Both" under each type of calls denote the number of all function calls, the calls that are only made by the authors of function definitions, the calls that are only made by developers other than the authors of definitions, and the calls that are made by both authors and other developers.

Index	Project	#Calls	#Calls of Type I					#Calls of Type II					#Calls of Type III				
			All	Self	Others	Both		All	Self	Others	Both		All	Self	Others	Both	
1	scala	5162	5047	376	1849	2822		103	44	38	21		12	4	8	0	
2	framework	266	234	95	32	107		32	5	6	21		0	0	0	0	
3	gitbucket	543	489	75	11	403		0	0	0	0		54	32	0	22	
4	finagle	204	186	90	45	51		16	15	1	0		2	2	0	0	
5	kafka	25	25	25	0	0		0	0	0	0		0	0	0	0	
6	lila	505	391	298	6	87		114	112	2	0		0	0	0	0	
7	bfg	5	3	2	1	0		1	1	0	0		1	1	0	0	
8	gatling	345	313	127	5	181		32	32	0	0		0	0	0	0	
9	scalaz	4191	3677	550	584	2543		438	54	66	318		76	7	35	34	
10	openwhisk	41	15	9	5	1		6	3	1	2		0	0	0	0	
11	sbt	1989	1415	295	201	919		505	76	38	391		69	13	11	45	
12	dotty	770	740	280	47	413		28	18	2	8		2	2	0	0	
13	scalding	846	829	201	75	553		13	10	3	0		4	0	4	0	
14	shapeless	46	36	20	4	12		10	4	6	0		0	0	0	0	
15	scalatra	33	21	18	3	0		12	1	2	9		0	0	0	0	
16	jobserver	30	24	16	8	0		6	4	2	0		0	0	0	0	
17	util	500	473	58	126	289		15	10	0	5		12	7	1	4	
18	slick	391	363	185	4	174		23	12	2	9		5	5	0	0	
19	lagom	35	34	23	2	9		1	1	0	0		0	0	0	0	
20	ammonite	133	125	64	1	60		8	8	0	0		0	0	0	0	
Total		16060	14440	2807	3009	8624		1363	410	169	784		237	73	59	105	

charts in `Type I`, the higher-order functions in 11 out of 20 projects are mostly called by their authors; in `Type II`, the higher-order functions in 13 projects are mostly called by their authors. However, in the total data of 20 projects, we find that the higher-order functions are mostly called by both authors and other developers. The reason for this observation is that the higher-order functions in three large projects, including `scala`, `scalaz`, and `sbt`, are mostly called by both authors and other developers. This indicate that in several large projects, such as Project `scala`, developers may tend to work more collaboratively than in small projects.

Among all definitions of higher-order functions in the study, inputing functions as parameters is the commonest type of defining higher-order functions. Among all calls of higher-order functions, most of calls are made by the same developers who have defined the functions; meanwhile, there indeed exist higher-order functions that are only called by developers other than their authors of definitions.

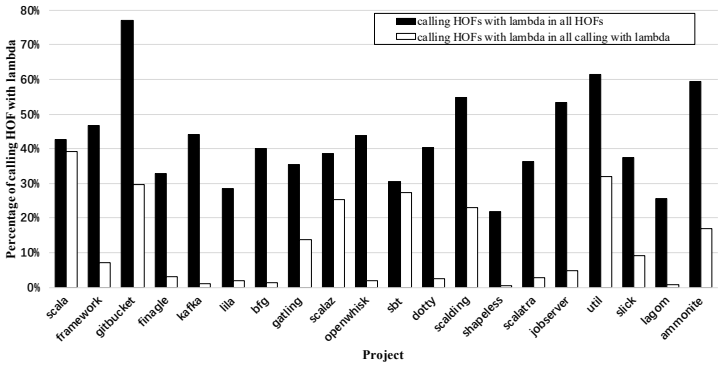

Fig. 12. Percentage of the overlap of calling higher-order functions and lambda expressions.

4.3 RQ3. How much is the overlap of calling higher-order functions and calling lambda expressions?

Goal. We aim to analyze the number of calling higher-order functions and lambda expressions and to show their overlap and difference.

A lambda expression is a kind of anonymous function without function identifier. In Scala language, a lambda expression uses the following syntax `(a: Int, b: Int) => a + b` to denote a function that adds two integers `a` and `b`. The lambda expressions have been drawn attention from the research community [10,2,32,17,21]. As a function, a lambda expressions can be called as a parameter of higher-order functions. The concepts of higher-order functions and lambda expressions are different: a higher-order function is a function of functions while a lambda expression is a kind of anonymous function. However, in practice, developers may feel confusing when facing higher-order functions and lambda expressions [30], [31]. In this research question, we investigate the calling of higher-order functions and lambda expressions in Scala programs.

We define two measures to count the overlap of calling higher-order functions and lambda expressions,

$$ratio(\frac{\#intersection}{\#higher\text{-}order}) = \frac{\#\text{ calls of higher-order functions that contain lambda expressions}}{\#\text{ calls of all higher-order functions}}$$

$$ratio(\frac{\#intersection}{\#lambda}) = \frac{\#\text{ calls of higher-order functions that contain lambda expressions}}{\#\text{ calls of all lambda expressions}}$$

Fig. 12 presents the ratio of overlap of calling higher-order functions and lambda expressions according above two measures. As shown in Fig. 12, 17 out of 20 projects have over 30% of lambda expressions in the calling of higher-order functions; 9 projects have over 40% of lambda expressions in the calling of higher-order functions, and 5 projects has over 50% of lambda expressions in the calling of higher-order functions. The illustration in Fig. 12 shows that

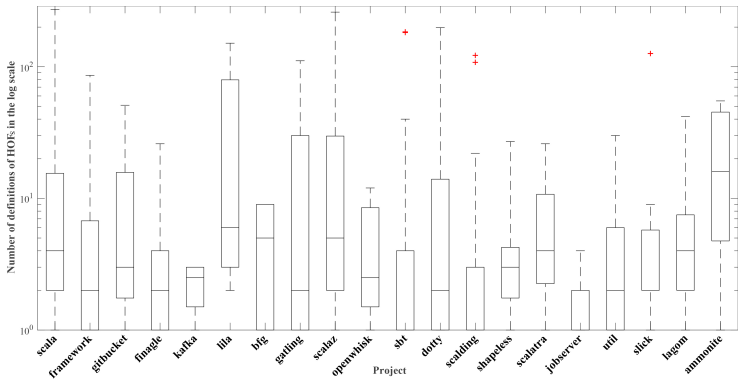

Fig. 13. Box-plots of function definitions per developer in the log scale.

calling lambda expressions does not indicate calling higher-order functions; vice versa. Meanwhile, the number of calling higher-order functions is higher than the number of calling lambda expressions in all 20 projects under in the study.

Our study shows that calling higher-order functions and lambda expressions behaves different in practice. This fact can be viewed as an empirical evidence that higher-order functions and lambda expressions should be carefully used in daily development.

4.4 RQ4. How do developers contribute to write higher-order functions?

Goal. Developers play an important role in defining and calling higher-order functions. We extracted and counted the number of defined or called functions that are contributed by different developers.

To understand the contribution of function definitions, we counted how many functions are defined by each developers. Fig. 13 presents the box-plots of function definitions per developer in the log scale. We can observe that developers behave different among projects. As shown in Fig. 13, the median value of function definitions by each developer is no more than 6 in 19 out of 20 projects; one exception is Project `ammonite`, whose median is 10.5. In 17 projects, the maximum value of contributed definitions by each developer is 10 or more; in 4 projects, the maximum value reaches over 100 definitions. In 14 projects, each of over 25% of developers have contributed 5 definitions of higher-order functions; in 8 projects, each of 25% of developers have contributed 10 definitions. We find out that several developers have indeed defined many higher-order functions in their daily development.

We also illustrated the contribution of calling higher-order functions. Fig. 14 presents the box-plots of function calls per developer in the log scale. The median value of contributed calls by each developer is 7 or less in 19 out of 20 projects; one exception is Project `shapeless` with the median of 10. In 15 out of 20

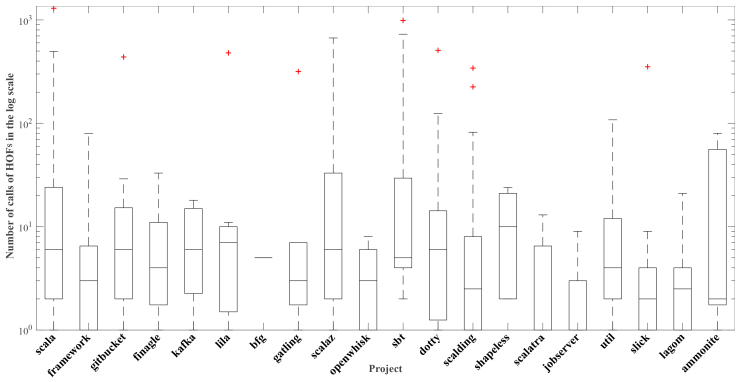

Fig. 14. Box-plots of function calls per developer in the log scale.

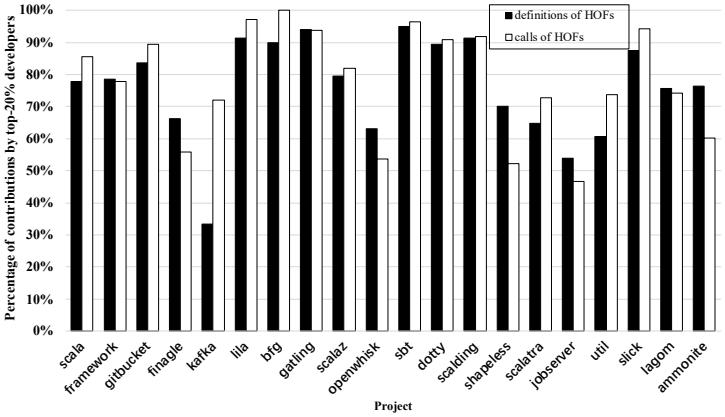

Fig. 15. Percentage of the contribution of definitions and calls of higher-order functions by top 20% developers.

projects, the maximum of contributed calls by one developer is over 10; in 5 projects, the maximum of calls is over 100. Meanwhile, in 11 projects, each of over 25% developers who have contributed 10 calls of higher-order functions; in 5 projects, each of over 25% developers who have 20 calls. This observation shows that there exist many developers who have called higher-order functions for multiple times.

To further understand the developer contribution, we examined the number of definitions and calls by the top 20% developers who have contributed the most. The choice of top 20% developers is derived from the 80-20 rule from the empirical study in sociology, which reveals that 80% of outcomes are made from 20% of products [12,25]. Fig. 15 presents the percentage of contributions by top 20% developers. Considering the contributions, on the one hand, 8 out of 20 projects have over 80% definitions of higher-order functions contributed by

Table 4. Spearman's rank correlation coefficient between the number of calls and three measurements (LoC, complexity, and #StyleWarnings) for each higher-order function.

	Min	Max	Average	Coefficient	p-value
LoC	1	104	4.5203	0.2301	<2.20E-16
Complexity	1	16	1.4051	0.1817	<2.20E-16
#StyleWarnings	0	29	0.4173	0.0016	0.9052

top 20% developers; 5 projects even have over 90% definitions of higher-order functions contributed by top 20% developers. On the other hand, 10 out of 20 projects have over 80% callings of higher-order functions, 7 projects have over 90% callings of higher-order functions contributed by top 20% developers. In total, the top 20% developers have contributed 76.15% function definitions and 78.06% calls.

Among 20 projects, the number of function definitions by each developer ranges from 1 to 351; the number of function calls by each developer ranges from 1 to 1297. The top 20% developers have contributed 76.15% function definitions and 78.06% calls in total.

4.5 RQ5. Which factor affects the calls of higher-order functions?

Goal. We tend to find out the factors that affect the number of calls of higher-order functions. In each project, we leverage the Spearman's rank correlation coefficient to show the correlation between the calls and LoC, Cyclomatic complexity, and #StyleWarnings, respectively.

We use the Spearman's rank correlation coefficient to measure the correlation between the number of calls and a factor that potentially affect the calls. Spearman's rank correlation coefficient is a non-parametric measure of the statistical correlation between two variables [34]. The correlation coefficient is calculated with the covariance of ranks of two given variables. The coefficient varies from -1 to 1. A positive coefficient means that a variable increases when the other variable increases while a negative coefficient means that a variable decreases when the other variable increases. The absolute value of the coefficient indicates the degree of correlation between two variables: zero is no correlation and one is completely correlated. We consider that a p-value less than 0.05 is statistically significant.

Table 4 presents the Spearman's rank correlation coefficient between the number of calls of higher-order functions and three defined measurements, LoC, Cyclomatic complexity, and #StyleWarnings. For each measurement, we show the minimum, the maximum, the average, the coefficient, and the p-value. As shown in the table, LoC and Complexity show the correlation coefficients of 0.23 and 0.18 with the number of function calls. That is, a higher-order function with more executable lines of code or higher complexity can be called for more times. A possible reason is that a function with many lines or high complexity may contain

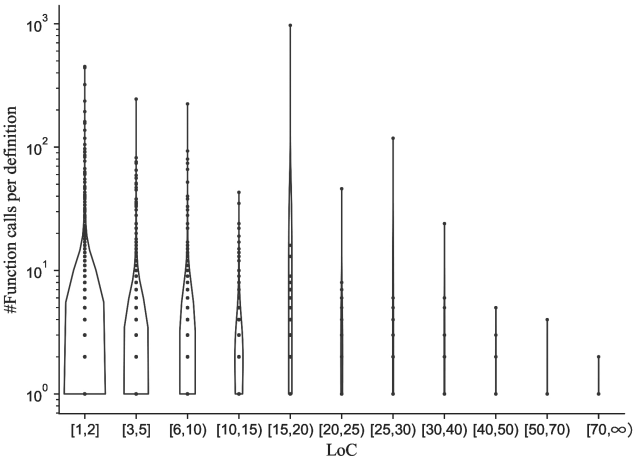

Fig. 16. Number of function calls per definition for LoC.

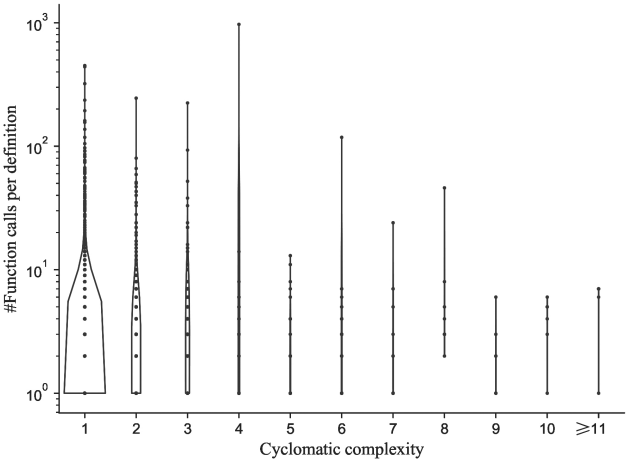

Fig. 17. Number of function calls per definition for Cyclomatic complexity.

rich information and detailed functionality. The number of reported warnings in the code style of a higher-order function does not show the correlation with the number of calls.

We illustrated the numbers of function calls for three measurements. Fig. 16 presents the illustration of number of function calls for different values of LoC. The width in each violin represents the density of function calls. As shown in Fig. 16, the shape of the violin gradually narrows when the LoC increases. Most of higher-order functions are called for one or twice. When the LoC is over 25, each violin basically behaves as a straight line; that is, the destiny is low. We

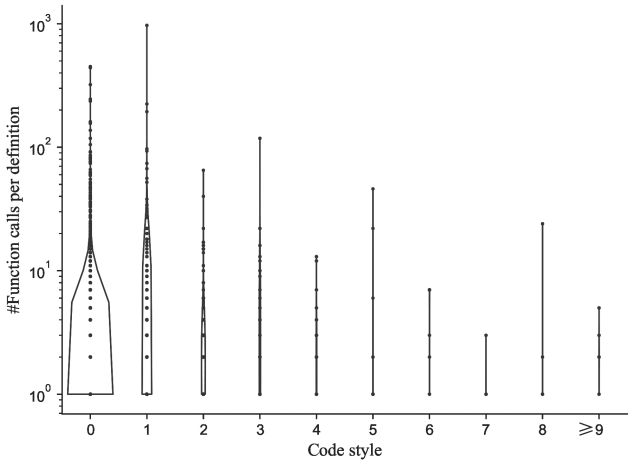

Fig. 18. Number of function calls per definition for #StyleWarnings.

can observe that the maximum number of calls shows a decreasing trend as the LoC increases, although there exists a slight fluctuation between the LoC of 10 and 30.

Fig. 17 illustrates the numbers of function calls for different values of Cyclomatic complexity of higher-order functions. We observe that the structure of most higher-order functions is not complicated and the higher-order functions with the Cyclomatic complexity of one account for the vast majority. As shown in Fig. 17, there exists a higher-order function with Cyclomatic complexity of four that has the most calls over 1000. As the Cyclomatic complexity of higher-order functions increases, the maximum number of calls of higher-order functions generally decreases. The maximum number of higher-order function calls with Cyclomatic complexity less than or equal to four is higher than the maximum number of higher-order function calls with Cyclomatic complexity over four.

Fig. 18 presents the illustration of the numbers of function calls with #StyleWarnings. As shown in Fig. 18, most higher-order functions have no code style warning. Higher-order functions with warnings concentrated on the number of one or two. Meanwhile, as #StyleWarnings increases, the maximum value of higher-order function calls fluctuates. One possible reason for this fact is that the requirements of the code style is not consistent among all projects in the study.

Table 5 presents the Spearman's rank correlation coefficient between the number of calls and the three measures, including LoC, Cyclomatic complexity, and #StyleWarnings, for higher-order functions in each project. As shown in Table 5, LoC and Cyclomatic complexity show positive correlations with the number of function calls in most projects; that is, a higher-order function with more executable lines of code or higher complexity can be called for more times. The p-values of correlation coefficients shows the statistical significance: the number of

Table 5. Spearman's rank correlation coefficient between the number of calls and the three measures (LoC, complexity, and #StyleWarnings) for each higher-order function in all projects under evaluation.

Project	LoC		Complexity		#StyleWarnings	
	Coefficient	p-value	Coefficient	p-value	Coefficient	p-value
scala	**0.2531**	<2.20E-16	**0.1684**	3.44E-09	**0.1162**	4.90E-05
framework	0.0515	0.4737	**-0.1757**	0.0138	0.0229	0.7502
gitbucket	-0.0449	0.7314	-0.0181	0.8898	0.0278	0.8318
finagle	**0.2426**	0.0022	**0.2122**	0.0076	0.0358	0.6565
kafka	0.6610	0.0526	0.2219	0.5661	0.2155	0.5776
lila	0.0823	0.2931	-0.1249	0.1099	**-0.2336**	0.0025
bfg	**0.3584**	0.0023	0.2162	0.0722	**0.2625**	0.0281
gatling	**0.2944**	0.0012	0.1551	0.0936	0.0577	0.5345
scalaz	**0.2489**	<2.20E-16	**0.2489**	<2.20E-16	**-0.1184**	3.18E-07
openwhisk	-0.3192	0.0508	-0.1820	0.2743	0.0489	0.7707
sbt	**0.1741**	0.0003	**0.1829**	0.0001	0.0061	0.8997
dotty	**0.2037**	0.0010	**0.1429**	0.0222	0.0648	0.3013
scalding	0.0397	0.4987	0.0730	0.2138	**-0.1933**	0.0009
shapeless	**0.3321**	0.0185	**0.4785**	0.0004	**0.3159**	0.0255
scalatra	**0.3976**	0.0022	0.2498	0.0610	**-0.3781**	0.0037
jobserver	-0.3588	0.2286	-0.4084	0.1659	-0.2307	0.4483
util	**0.2791**	0.0002	**0.3296**	7.96E-06	-0.0802	0.2899
slick	0.0758	0.3395	**0.1702**	0.0309	-0.0166	0.8341
lagom	**0.3584**	0.0023	0.2162	0.0722	**0.2625**	0.0281
ammonite	**0.3932**	0.0006	0.1805	0.1291	-0.0611	0.6103

calls of higher-order functions correlates with the LoC in 12 out of 20 projects, with the Cyclomatic complexity in 9 projects, and with #StyleWarnings in 8 projects. For the LoC, coefficients of 12 projects with statistical significance have positive correlations of over 0.17. For the Cyclomatic complexity, coefficients of 8 out of 9 projects with statistical significance have positive correlations. For #StyleWarnings, coefficients of 4 out of 8 projects with statistical significance show positive correlations with the number of function calls while coefficients of the other 4 projects show negative correlations.

The number of calling higher-order functions correlates with the number of executable lines of code (with an coefficient of 0.23) and the Cyclomatic complexity (with an coefficient of 0.18). Results on individual projects show that the correlations with the number of executable lines of code are positive; all correlations but one with the Cyclomatic complexity are positive; the warnings of the code style contain both positive correlations and negative correlations.

5 Threats to Validity

We discuss the threats to the validity to our work in three dimensions.

Threats to construct validity. In the study, we quantified the complexity and the warnings in the code style of a higher-order function via leveraging

three measures, i.e., LoC, Cyclomatic complexity, and #StyleWarnings. We chose these three measures because they are widely used and can be simply extracted via off-the-shelf tools. We notice that there exist several other measures or tools that can be used as measures, such as the Halstead complexity to measure the functional complexity based on parsing operators [1]. We plan to involve other measures in further work. In Section 3.1, we identify developers via their e-mail addresses. However, if a developer uses two or more e-mail addresses, it is difficult to simultaneously match these e-mail addresses to the same developer. Such multiple e-mail addresses of a single developer may hurt the computation of developer contributions, e.g., the result in Section 4.4.

Threats to internal validity. The correlation analysis in the paper may be biased by potential confounding variables. Since it is difficult to exhaust many potential variables, we used Spearman's rank correlation coefficient to measure the linear correlation between two variables. The experimental results in the paper can be viewed as the observation and could be further explored.

Threats to external validity. Our study selected 20 Scala projects according the number of stars from GitHub. Therefore, our empirical study may not represent the general result of using higher-order functions in all Scala projects. Selecting projects based on the number of stars or the number of forks may lead to the bias of sampled projects. In our study, we have filtered out several projects, such as Project `scala-js` due to the issues of configurations and requirements of the tool SemanticDB. Such filtering may also result in the selection bias of projects. The experimental results can only indicate the observation and findings based on the data collection and preparation in this paper

6 Related Work

The aim of this paper is to conduct an exploratory study on using higher-order functions in Scala programs. We summarized the related work in two categories, the study on using higher-order functions and the study on Scala programs.

6.1 On Using Higher-Order Functions

Many studies used higher-order functions as a new paradigm to solve complex problems. Wester and Kuper [35] applied higher-order functions as a trade-off between time and area for large digital signal processing applications. They further converted the higher-order functions in Haskell into data flow nodes to weigh particle filter time and space consumption [36]. Clark and Barn [8] used higher-order functions in dynamic reconstruction of event-driven architectures to increase the flexibility of the model. Bassoy and Schatz [4] optimized higher-order functions to quickly calculate tensors; their optimized implementation achieved 68% of the maximum throughput of the Intel Core i9-7900X. Nakaguchi et al. [20] treated services as functions and used higher-order functions to combine these services without creating new services. Racordon [24] leveraged higher-order functions to

implement components to provide coroutines for programming language without coroutines.

Existing studies have been conducted to understand the difficulty of verifying and testing higher-order functions. Madhavan et al. [16] presented a novel approach that uses lazy evaluation and memoization to specify and verify the resource utilization of higher-order functional programs. Voirol et al. [33] presented a validator for pure higher-order functional Scala programs, which support arbitrary function types and arbitrary nested anonymous functions. Rusu and Arusoaie [28] embedded a higher-order functional language with imperative features into the Maude framework to verify higher-order functional programs. Selakovic et al. [29] presented LambdaTester to automate test generation for higher-order functions in dynamic languages. Lincke and Schupp [14] proposed the transformation that converts higher-order functions into lower-order functions by mapping higher-order types to lower-order types.

In this paper, we proposed the first study on how developers use higher-order functions in Scala programs. We conducted five research questions to understand the definitions and calls of higher-order functions.

6.2 On Scala Programs

The Scala language has been widely studied in the research community. We list several related works to briefly introduce the recent progress on the study of Scala programs. Cassez and Sloane [7] presented a Scala library called ScalaSMT, which supports the Satisfiability Modulo Theory (SMT) solving in Scala via accessing mainstream SMT solvers. Kroll et al. [13] used pattern matching of the Scala language and presented a framework that supports the straightforward and simplified translation via connecting a formal algorithm specification and executable code. To implement efficient super-compilers for arbitrary programming languages, Nystrom [22] designed a Scala framework that can be used for experimenting with super-compilation techniques and constructed directly from an interpreter. Reynders et al. [26] defined a multi-tier language, Scalagna, which combines the existing Scala JVM and JavaScript ecosystems into a single programming model without requiring changes or rewrites of existing Scala compilers. Karlsson and Haller [11] presented the first implemented design for records in Scala which enables type-safe record operations. In the field of eduction, van der Lippe et al. [15] leveraged the Scala programming language and the WebLab online learning management system to automate specification tests on the submissions by students submissions. Additionally, they have developed a scalable solution for running a course on concepts of programming languages using definitional interpreters.

Our study in this paper focuses on higher-order functions, an important feature of the Scala language. Understanding the use of higher-order functions can help improve the reusability and maintenance of source code.

7 Conclusions

In this paper, we conducted an exploratory study on the use of higher-order functions in Scala programs. We collected definitions, calls, and authors of higher-order functions from 20 Scala projects with the most stars. Our study shows that higher-order functions are not widely used in Scala programs and most of higher-order functions are defined and called by the same developers. We also find that the number of calls of higher-order functions correlates most with the number of executable lines of code and the code complexity. Constructing higher-order functions from existing functions decreases the number of similar functions and improves the code reuse; understanding higher-order functions can be used to support automated code reuse and code generation.

In future work, we plan to conduct user questionnaires to invite developers to further evaluate the use of higher-order functions. Such evaluation is to reveal potential difficulty or practical issues in using higher-order functions. We also plan to conduct experiments on the changes on higher-order functions, e.g., the different developers who have changed the definition of higher-order functions. This may help understand the evolution of higher-order functions and guide the future development with higher-order functions. Another future work is to conduct automated code reuse and refactoring via abstracting higher-order functions from existing functions.

Acknowledgments. This work was supported by the National Natural Science Foundation of China (Grant No. 61872273).

References

1. Albrecht, A.J., Jr., J.E.G.: Software function, source lines of code, and development effort prediction: A software science validation. IEEE Trans. Software Eng. **9**(6), 639–648 (1983)
2. Arefin, M., Khatchadourian, R.: Porting the netbeans java 8 enhanced for loop lambda expression refactoring to eclipse. In: Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity, SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015. pp. 58–59 (2015)
3. Bacchelli, A., Bird, C.: Expectations, outcomes, and challenges of modern code review. In: Proceedings of the 2013 International Conference on Software Engineering. pp. 712–721. IEEE Press (2013)
4. Bassoy, C., Schatz, V.: Fast higher-order functions for tensor calculus with tensors and subtensors. In: Computational Science - ICCS 2018 - 18th International Conference, Wuxi, China, June 11-13, 2018, Proceedings, Part I. pp. 639–652 (2018)
5. Brachthäuser, J.I., Schuster, P.: Effekt: extensible algebraic effects in scala. In: Proceedings of the 8th ACM SIGPLAN International Symposium on Scala, SCALA@SPLASH 2017, Vancouver, BC, Canada, October 22-23, 2017. pp. 67–72 (2017)
6. Cardelli, L., Wegner, P.: On understanding types, data abstraction, and polymorphism. ACM Comput. Surv. **17**(4), 471–522 (1985)

7. Cassez, F., Sloane, A.M.: Scalasmt: satisfiability modulo theory in scala. In: Proceedings of the 8th ACM SIGPLAN International Symposium on Scala, SCALA@SPLASH 2017, Vancouver, BC, Canada, October 22-23, 2017. pp. 51–55 (2017)
8. Clark, T., Barn, B.S.: Dynamic reconfiguration of event driven architecture using reflection and higher-order functions. Int. J. Software and Informatics **7**(2), 137–168 (2013)
9. Gousios, G., Storey, M.D., Bacchelli, A.: Work practices and challenges in pull-based development: the contributor's perspective. In: Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22. pp. 285–296 (2016)
10. Järvi, J., Freeman, J.: C++ lambda expressions and closures. Sci. Comput. Program. **75**(9), 762–772 (2010)
11. Karlsson, O., Haller, P.: Extending scala with records: design, implementation, and evaluation. In: Proceedings of the 9th ACM SIGPLAN International Symposium on Scala, SCALA@ICFP 2018, St. Louis, MO, USA, September 28, 2018. pp. 72–82 (2018)
12. Koch, R.: The 80/20 Principle: The Secret of Achieving More with Less: Updated 20th anniversary edition of the productivity and business classic. Hachette UK (2011)
13. Kroll, L., Carbone, P., Haridi, S.: Kompics scala: narrowing the gap between algorithmic specification and executable code (short paper). In: Proceedings of the 8th ACM SIGPLAN International Symposium on Scala, SCALA@SPLASH 2017, Vancouver, BC, Canada, October 22-23, 2017. pp. 73–77 (2017)
14. Lincke, D., Schupp, S.: From HOT to COOL: transforming higher-order typed languages to concept-constrained object-oriented languages. In: International Workshop on Language Descriptions, Tools, and Applications, LDTA '12, Tallinn, Estonia, March 31 - April 1, 2012. p. 3 (2012)
15. van der Lippe, T., Smith, T., Pelsmaeker, D., Visser, E.: A scalable infrastructure for teaching concepts of programming languages in scala with weblab: an experience report. In: Proceedings of the 7th ACM SIGPLAN Symposium on Scala, SCALA@SPLASH 2016, Amsterdam, Netherlands, October 30 - November 4, 2016. pp. 65–74 (2016)
16. Madhavan, R., Kulal, S., Kuncak, V.: Contract-based resource verification for higher-order functions with memoization. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017. pp. 330–343 (2017)
17. Mazinanian, D., Ketkar, A., Tsantalis, N., Dig, D.: Understanding the use of lambda expressions in java. PACMPL **1**(OOPSLA), 85:1–85:31 (2017)
18. McCabe, T.J.: A complexity measure. IEEE Trans. Software Eng. **2**(4), 308–320 (1976)
19. McIntosh, S., Kamei, Y., Adams, B., Hassan, A.E.: An empirical study of the impact of modern code review practices on software quality. Empirical Software Engineering **21**(5), 2146–2189 (2016)
20. Nakaguchi, T., Murakami, Y., Lin, D., Ishida, T.: Higher-order functions for modeling hierarchical service bindings. In: IEEE International Conference on Services Computing, SCC 2016, San Francisco, CA, USA, June 27 - July 2, 2016. pp. 798–803 (2016)
21. Nielebock, S., Heumüller, R., Ortmeier, F.: Programmers do not favor lambda expressions for concurrent object-oriented code. Empirical Software Engineering **24**(1), 103–138 (2019)

22. Nystrom, N.: A scala framework for supercompilation. In: Proceedings of the 8th ACM SIGPLAN International Symposium on Scala, SCALA@SPLASH 2017, Vancouver, BC, Canada, October 22-23, 2017. pp. 18–28 (2017)
23. Odersky, M., Altherr, P., Cremet, V., Emir, B., Maneth, S., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., Zenger, M.: An overview of the scala programming language. Tech. rep., Technical Report IC/2004/64, EPFL Lausanne, Switzerland (2004)
24. Racordon, D.: Coroutines with higher order functions. CoRR **abs/1812.08278** (2018)
25. Reed, W.J.: The pareto, zipf and other power laws. Economics letters **74**(1), 15–19 (2001)
26. Reynders, B., Greefs, M., Devriese, D., Piessens, F.: Scalagna 0.1: towards multi-tier programming with scala and scala.js. In: Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming, Nice, France, April 09-12, 2018. pp. 69–74 (2018)
27. Richmond, D., Althoff, A., Kastner, R.: Synthesizable higher-order functions for C++. IEEE Trans. on CAD of Integrated Circuits and Systems **37**(11), 2835–2844 (2018)
28. Rusu, V., Arusoaie, A.: Executing and verifying higher-order functional-imperative programs in maude. J. Log. Algebr. Meth. Program. **93**, 68–91 (2017)
29. Selakovic, M., Pradel, M., Karim, R., Tip, F.: Test generation for higher-order functions in dynamic languages. PACMPL **2**(OOPSLA), 161:1–161:27 (2018)
30. Stackoverflow: Is lambda a type of higher-order function? `http://stackoverflow.com/questions/4999533/is-lambda-a-type-of-higher-order-function` (2019)
31. Stackoverflow: Lambda expressions and higher-order functions. `http://stackoverflow.com/questions/15198979/lambda-expressions-and-higher-order-functions` (2019)
32. Tsantalis, N., Mazinanian, D., Rostami, S.: Clone refactoring with lambda expressions. In: Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017. pp. 60–70 (2017)
33. Voirol, N., Kneuss, E., Kuncak, V.: Counter-example complete verification for higher-order functions. In: Proceedings of the 6th ACM SIGPLAN Symposium on Scala, Scala@PLDI 2015, Portland, OR, USA, June 15-17, 2015. pp. 18–29 (2015)
34. Walpole, R.E., Myers, S.L., Ye, K., Myers, R.H.: Probability and statistics for engineers and scientists. Pearson (2007)
35. Wester, R., Kuper, J.: A space/time tradeoff methodology using higher-order functions. In: 23rd International Conference on Field programmable Logic and Applications, FPL 2013, Porto, Portugal, September 2-4, 2013. pp. 1–2 (2013)
36. Wester, R., Kuper, J.: Design space exploration of a particle filter using higher-order functions. In: Reconfigurable Computing: Architectures, Tools, and Applications - 10th International Symposium, ARC 2014, Vilamoura, Portugal, April 14-16, 2014. Proceedings. pp. 219–226 (2014)
37. Zhang, X., Chen, Y., Gu, Y., Zou, W., Xie, X., Jia, X., Xuan, J.: How do multiple pull requests change the same code: A study of competing pull requests in github. In: 2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018. pp. 228–239 (2018)
38. Zou, W., Xuan, J., Xie, X., Chen, Z., Xu, B.: How does code style inconsistency affect pull request integration? an exploratory study on 117 github projects. Empirical Software Engineering (Jun 2019)