Empirical Software Engineering (2020) 25:4547–4584
https://doi.org/10.1007/s10664-020-09842-7

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] · Jifeng Xuan[1]

Published online: 4 September 2020

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 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 8,285 higher-order functions from 35 Scala projects in 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 definition types, the definition distribution, the factor that correlates with the function calls, and the developer contribution. Our study mainly shows five empirical results about the common use of higher-order functions in Scala programs. Our findings are listed as follows. (1) Among 35 Scala projects, 6.84% of functions are defined as higher-order functions on average and the average calls per function show that higher-order functions are called more frequently than first-order functions. (2) In all higher-order functions in the study, 87.35% of definitions of higher-order functions and 90.66% of calls belong to the type that only takes functions as parameters. (3) Three measurements (including lines of executable code, Cyclomatic complexity, and warnings in the code style) in higher-order functions are lower than those of first-order functions. (4) Regression analysis on all projects suggests that the number of calling higher-order functions highly correlates with the Cyclomatic complexity. (5) In all projects in the study, 43.82% calls of higher-order functions are written by the same developers who have defined the functions and results show that top 20% authors of higher-order functions favor defining or calling higher-order functions than first-order functions. This study can be viewed as a preliminary result to understand the use of higher-order functions and to motivate further investigation in Scala programs.

This article belongs to the Topical Collection: *Software Applications (NASAC)*

Communicated by: Richard Paige, Jordi Cabot and Neil Ernst

✉ Jifeng Xuan
jxuan@whu.edu.cn

Extended author information available on the last page of the article.

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 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 reducing the search space of potential functions.

Many program languages support programming with higher-order functions, such as C++ 11, Java 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 prevents higher-order functions from being widely used.

Developers employ higher-order functions to enhance the design of source code in the 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 returning functions as the output (Richmond et al. 2018). 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 (Cardelli and Wegner 1985). 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.[1] Third, for complex calculations like high-precision control systems, developers can use higher-order functions to keep the source code of calculations simple and clear. For example, Bassoy and Schatz (2018) reported that the tensor data with a non-hierarchical storage format and an arbitrary number of dimensions can be handled by the recursive multi-index algorithms implemented in higher-order functions.

1.1 Background

Among programming languages that support higher-order functions, we focus on the Scala language in this paper. Scala is a multi-paradigm programming language that provides the support of functional programming and a strong static type system (Odersky et al. 2004). We chose to study higher-order functions in Scala programs due to the following two reasons. On the one hand, different from purely functional programming languages like Standard ML and Haskell, Scala combines object-oriented and functional programming into one high-level language (Scala 2020). The object-oriented features make Scala widely used in different domains (Nystrom 2017; Kroll et al. 2017). For example, Twitter decided to migrate its back-end programs from Ruby (an object-oriented programming language) to

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

Scala (HackerNews 2009). On the other hand, different from dynamic programming languages like Python, Scala is a static type-system language like Java. The static type system enables static and concise program analysis of Scala programs. The Scala code can be compiled into Java bytecode. This design enables the compiled code to be directly executed on the Java virtual machine. Using higher-order functions in Scala can improve the abstraction and simplification of function design, which is not originally supported in Java.[2]

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 (Karlsson and Haller 2018). Such design makes the Scala language widely used in many development scenarios, e.g., the rapid web development and the construction of distributed systems. For instance, Nystrom (Nystrom 2017) has presented a Scala framework for experimenting with super-compilation techniques; Kroll et al. (2017) have proposed the Scala platform that conducts a straightforward and simplified translation from a formal specification to source code.

Scala shares many features of functional programming languages, including currying, type inference, and immutability. 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 decrease readability (Selakovic et al. 2018). For instance, if a higher-order function returns a function, the type of the returned function can be omitted from the definition of this higher-order function.

Figure 1 shows an excerpt of a real-world higher-order function `readModifiers()` in Project `lampepfl/dotty`.[3] 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 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. The parameters are `end`, `readAnnot`, `readWithin`, `defaultWithin`, and `ctx`. Among these five parameters, two parameters, `end` at Line 2 and `defaultWithin` at Line 5, 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`; 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.[4] As shown in Fig. 1, the second parameter `readAnnot` at Line 3 of the function `readModifiers()` is a higher-order function, which receives a

[2]Java supports higher-order functions since its Version 8.0 in 2014.

[3]Project dotty, http://github.com/lampepfl/dotty.

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

```
 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

`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` 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.

1.2 Findings and Contributions

Higher-order functions have been applied in the development of many applications (Wester and Kuper 2013; Brachthäuser and Schuster 2017; Nystrom 2017). However, there is no prior study that investigates the use of higher-order functions. For instance, in a mature project, how many functions are higher-order functions? Do developers use higher-order functions in the same way as the other functions? Who has defined or called the higher-order function? Which factor correlates with the number of function calls? In this paper, we extracted data of 35 popular Scala projects with the most stars from GitHub and conducted an exploratory study on understanding the use of higher-order functions in Scala programs.

We leveraged static analysis to extract 8,285 definitions and 22,579 calls of higher-order functions by 521 developers from 1,326.3k executable lines of Scala code. Our study is to answer five research questions.

- **RQ1. How many higher-order functions are there in Scala projects?** We show basic statistics on the higher-order functions in 35 Scala projects and show the existence of higher-order functions. Among 35 Scala projects in our study, 6.84% of functions are defined as higher-order functions in average while higher-order functions are called more frequently than first-order functions according to the average calls per function.
- **RQ2. How are higher-order functions defined?** We briefly categorize the definitions of higher-order functions according to whether the argument list or the returned value contains functions. Higher-order functions that take functions as parameters are the most common type of definitions. A higher-order function that takes at least one function as a parameter and returns functions is not frequently defined and called.
- **RQ3. How do definitions of higher-order functions distribute?** We quantify definitions of higher-order functions with three measurements, including LoC, complexity, and warnings in code style. The average and the standard deviation values show that the measurement values of higher-order functions are lower than those of first-order functions.
- **RQ4. Which factor correlates with the calls of higher-order functions?** We further analyze the factors that correlate with the calls of higher-order functions. We build multivariate linear regression model between factors and the number of function calls of higher-order functions. The analysis suggests that the number of calls for a higher-order function highly correlates with the Cyclomatic complexity. Results on individual projects show that correlations with the number of executable lines of code are positive in 29 projects; all correlations but one with the Cyclomatic complexity are positive.
- **RQ5. How do developers contribute to defining and calling higher-order functions?** We analyze the number of higher-order functions defined and called by each developer. Among all calls of higher-order functions, 9894 (43.82%) calls are made by the same developers who have defined the functions. Results suggest that top 20% authors of higher-order functions favor defining or calling higher-order functions than first-order functions.

This paper makes the following major contributions:

1. We mined 35 Scala projects with the most stars in GitHub and collected 8,285 higher-order functions, which are contributed by 521 developers.
2. We empirically investigated the use of higher-order functions in Scala programs and designed an exploratory study on using higher-order functions via answering five research questions, including the data scale, the definition types, the definition distribution, the factor that correlates with the function calls, and the developer contribution.

The rest of this paper is organized as follows. Section 2 presents the study setup, including five research questions and the data preparation. Section 3 describes the results of our exploratory study. Section 4 discusses the threats to the validity. Section 5 lists the related work and Section 6 concludes.

2 Study Setup

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

2.1 Data Preparation

Our study aims to understand the use of higher-order functions in Scala programs. We mined 35 Scala projects and extracted data for further analysis, including function definitions, calls, developers who have written the functions, and the function complexity.[5] 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.[6] The main steps of data preparation are listed as follows.

Project selection We sorted all Scala projects in GitHub according to the stars.[7] A project with many stars indicates that the project is favored by developers because of the usage and quality. We selected top-50 projects with the most stars. Since applying SemanticDB to a project requires the compatible configuration of the project, we skipped 15 projects that cannot be parsed by SemanticDB and kept the other 35 projects. In detail, 15 projects were skipped: since SemanticDB can only support static analysis of Scala 2.11 or Scala 2.12, seven projects that are implemented in Scala 2.10 were skipped, including Projects `scala-js/scala-js`, `scala-native/scala-native`, `scalanlp/breeze`, `spray/spray`, `pocorall/scaloid`, `monix/moinx`, and `coursier/coursier`; Project `typelevel/spire` was also skipped since the implementation of this project is Scala 2.13; another five projects, `apache/spark`, `intel-analytics/BigDL`, `safeforce/TransmogrifAI`, `datastax/spark-cassandra-connector`, and `GravityLabs/goose` that are mixedly implemented with Scala and other programming languages are skipped due to the failure of configuring with SemanticDB; we also removed two projects `fpinscala/fpinscala` (a supplement material of practices in a book) and `scala-exercises/scala-exercises` (exercises for many libraries of Scala), since these projects are not real software projects. Table 1 lists a summary of 35 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,

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

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

[7]Scala projects with stars, http://github.com/search?l=Scala&o=desc&q=scala&s=stars&type=Repositories, accessed on September 1st, 2019.

Table 1 Summary of 35 Scala projects in GitHub in the study

Project	Abbr.	#Star	LoC	Project description
scala/scala	scala	12.1k	143.6k	The Scala programming language
apache/predictionio	predictionio	12.1k	20.3k	A machine learning framework
playframework/playframework	framework	11.3k	41.5k	A web framework for building scalable applications with Java and Scala
akka/akka	akka	10.3k	114.8k	A tool for building concurrent and distributed applications
yahoo/kafka-manager	kafka	8.0k	10.5k	A tool for managing Apache KafKa
gitbucket/gitbucket	gitbucket	7.8k	19.1k	A Git platform powered by Scala
twitter/finagle	finagle	7.3k	63.0k	An extensible RPC system for the Java JVM
ornicar/lila	lila	5.8k	70.3k	A free server for the online chess game
rtyley/bfg-repo-cleaner	bfg	5.7k	1.5k	A simple and fast tool for cleansing bad data out of Git repository
gatling/gatling	gatling	4.4k	25.6k	A highly capable load testing tool
scalaz/scalaz	scalaz	4.2k	35.4k	A Scala library for functional programming
sbt/sbt	sbt	4.0k	34.9k	A build tool for Scala, Java, and other languages
lampepfl/dotty	dotty	3.6k	387.1k	A Scala compiler
twitter/scalding	scalding	3.2k	29.6k	A Scala API for a Java tool named cascading
milessabin/shapeless	shapeless	2.9k	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	2.4k	7.6k	A REST job server for Apache Spark
twitter/util	util	2.3k	27.4k	A collection of core JVM libraries of Twitter
slick/slick	slick	2.3k	20.8k	A Scala library for database querying and accessing
lagom/lagom	lagom	2.3k	21.8k	A framework for building reactive microservice systems in Java or Scala
lihaoyi/Ammonite	ammonite	2.1k	8.6k	A Scala tool for scripting purposes
twitter/finatra	finatra	1.9k	20.2k	A framework for building applications on TwitterServer and Finagle

Table 1 (continued)

Project	Abbr.	#Star	LoC	Project description
twitter/algebird	algebird	1.9k	22.7k	A library for abstracting algebra in Scala
circe/circe	circe	1.8k	6.9	A JSON library for Scala
Azure/mmlspark	mmlspark	1.7k	18.0k	Microsoft machine learning for Apache Spark
http4s/http4s	http4s	1.7k	29.1k	A minimal, idiomatic Scala interface for HTTP services
spotify/scio	scio	1.7k	30.5k	A Scala API for Apache Beam and Google Cloud Dataflow
sangria-graphql/sangria	sangria	1.6k	15.0k	A Scala library for the GraphQL data query language
typelevel/scalacheck	scalacheck	1.6k	3.4k	A library for testing Scala or Java programs
zio/zio	zio	1.6k	13.9k	A Scala library for asynchronous and concurrent programming
getquill/quill	quill	1.5k	15.6k	A compile-time language integrated queries for Scala
tpolecat/doobie	doobie	1.5k	14.7k	A pure functional JDBC layer for Scala
functional-streams-for-scala/fs2	fs2	1.5k	9.2k	A Scala library for functional streams
foundweekends/giter8	giter8	1.5k	1.4k	A command line tool for applying templates published on Github
ThoughtWorksInc/Binding.scala	binding	1.4k	3.4k	A data-binding framework for Scala
Total		137.0k	1326.3k	

For the sake of space, each project will be denoted by its abbreviation in the following sections

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. In all 35 projects in Table 1, there are 15,239 Scala files and 1,326.3k executable lines of code in total; 8,285 definitions of higher-order functions with 22,579 calls are recorded for further analysis.

Developer extraction We mined the information about 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 traced back all logged changes. We 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 authors of the higher-order function according to the number of changes the author made for this higher-order function and identified the developer who wrote most changes as its original author (Zhang et al. 2018); similarly, for the source code of a function call, we identified the developer who wrote most changes of this function call (adding or modifying the function call) as its original author. If two or more developers wrote the same number of changes, the developer who wrote the earliest change among these developers is identified as the author.

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 represent the size of a function.

Code complexity measurement We use the Cyclomatic complexity to measure the complexity of the definition of a higher-order function. *Cyclomatic complexity* is a software metric of linearly independent paths (McCabe 1976); to measure the Cyclomatic complexity, we used the static analysis tool Scalameta to parse Scala files and construct the ASTs.[8] Then we identified the Cyclomatic complexity of each higher-order function by traversing its AST. For each function, we constructed its control flow graph by traversing the abstract syntax tree. A control flow graph is an abstract representation based on predicate nodes, which consists of all paths in the program execution (Gu et al. 2019). In a control flow graph, a node represents one or more consecutive unbranched statements and an edge represents a branch between these nodes. A predicate node is a node that represents a predicate in an `if`-condition or a loop. Cyclomatic complexity is defined as the number of predicate nodes plus one. This definition is equivalent to $e - n + 2$, where e and n are the number of edges and the number of nodes.

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 *#Style-Warnings* as the number of suspected issues of code style via an off-the-shelf tool of code style checking, ScalaStyle.[9] The ScalaStyle tool can extract 40 types of code style issues that relate to function code, including the existence of braces of `if`-statements, the use of magic numbers, the redundant whitespace after left brackets. For the definition of each higher-order function, we count the number of issues inside the definition based on the result of running ScalaStyle.

2.2 Research Questions

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

RQ1. How many higher-order functions are there 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 prevalence of

[8]Scalameta, http://scalameta.org/.

[9]ScalaStyle, http://www.scalastyle.org/.

higher-order function, i.e., how many higher-order functions are there in Scala programs. We analyze the definitions and calls of higher-order functions in RQ1.

RQ2. How are higher-order functions defined? In general, a higher-order function can take a function as a parameter and/or output a function as a returned result. This leads to three categories of higher-order functions based on the input and the output. We intend to investigate the definition types of higher-order functions in RQ2.

RQ3. How do definitions of higher-order functions distribute? The definition of a higher-order function can be characterized by code measurements. We analyze the distributions of function definitions of higher-order functions and other functions in RQ3.

RQ4. Which factor correlates with the calls of higher-order functions? Higher-order functions are expected to abstract the use pattern of functions (Karlsson and Haller 2018). In RQ4, we investigate the potential factors that correlate with the number of calls of higher-order functions. Empirical results of RQ4 can provide a way to understand the correlation between calls for higher-order functions and factors of definitions.

RQ5. How do developers contribute to defining and calling higher-order functions? The source code of higher-order functions is designed and written by developers. In RQ5, we aim to examine how many developers have contributed to defining and calling higher-order functions.

3 Empirical Results

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

3.1 RQ1. How Many Higher-Order Functions are there in Scala Projects?

Method We answer RQ1 via exploring the scale of using higher-order functions. We count definitions and calls of higher-order functions in each Scala project. We compare the average number of function calls of higher-order functions and other functions.

Result and analysis We collected definitions and calls of all higher-order functions in 35 Scala projects. Besides higher-order functions, we collected the non-higher-order functions, called *first-order functions* (Altenkirch 2001). Table 2 lists the numbers of definitions and calls of higher-order functions and first-order functions. In total, there are 8,285 definitions and 22,579 calls of these higher-order functions. Among 35 projects, higher-order functions account for 6.84% of function definitions. The percentage of higher-order functions varies largely across 35 projects: the ratio of definitions of higher-order functions ranges from 1.50% to 24.82%. In twenty projects, over 5% of functions are defined as higher-order functions. Among 35 projects, Project `zio` reaches the highest ratio of defining higher-order functions and Project `kafka` reaches the lowest ratio of definitions. There are 14 out of 35 projects, whose ratio of higher-order functions among all function definitions are higher than the average ratio 6.84% when we consider the ratio of the definitions of higher-order functions among all functions.

Table 2 Numbers of definitions and calls of higher-order functions and first-order functions in 35 Scala projects

Abbr.	Definitions			Calls of HOFs		Calls of FOFs	
	#All	#HOFs	Ratio (%)	#Total calls	#Avg. calls	#Total calls	#Avg. calls
scala	24731	1216	4.92%	5162	**4.25**	90840	3.86
predictionio	548	24	4.38%	52	**2.17**	1029	1.96
framework	3079	196	6.37%	266	1.36	5208	**1.81**
akka	13654	519	3.80%	1292	**2.49**	29915	2.28
kafka	600	9	1.50%	25	**2.78**	1191	2.02
gitbucket	1046	61	5.83%	543	**8.90**	2675	2.72
finagle	6147	157	2.55%	204	**1.30**	5700	0.95
lila	5715	165	2.89%	505	**3.06**	10735	1.93
bfg	116	10	8.62%	5	0.50	139	**1.31**
gatling	2690	118	4.39%	345	**2.92**	3841	1.49
scalaz	8144	1853	22.75%	4191	**2.26**	10403	1.65
sbt	4043	436	10.78%	1989	**4.56**	11649	3.23
dotty	10349	256	2.47%	770	3.01	40542	**4.02**
scalding	4152	292	7.03%	846	**2.90**	5439	1.41
shapeless	1934	50	2.59%	46	0.92	2370	**1.26**
scalatra	1477	57	3.86%	33	0.58	1790	**1.26**
jobserver	619	13	2.10%	30	**2.31**	668	1.10
util	2760	176	6.38%	500	**2.84**	2778	1.08
slick	2881	161	5.59%	391	**2.43**	3684	1.35
lagom	1805	70	3.88%	35	0.50	1362	**0.79**
ammonite	792	72	9.09%	133	**1.85**	1078	1.50
finatra	657	38	5.78%	14	0.37	576	**0.93**
algebird	2228	230	10.32%	176	0.77	4228	**2.12**
circe	944	113	11.97%	68	0.60	710	**0.85**
mmlspark	1902	60	3.15%	87	**1.45**	1873	1.02
http4s	2474	160	6.47%	273	**1.71**	3493	1.51
scio	2120	156	7.36%	435	**2.79**	3181	1.62
sangria	1682	161	9.57%	133	0.83	3449	**2.27**
scalacheck	454	100	22.03%	239	**2.39**	812	2.29
zio	2603	646	24.82%	1524	**2.36**	4033	2.06
quill	1414	123	8.70%	213	1.73	3192	**2.47**
doobie	5672	426	7.51%	1485	**3.49**	2305	0.44
fs2	1442	150	10.40%	545	**3.63**	2380	1.84
giter8	123	5	4.07%	4	0.80	122	**1.03**
binding	184	6	3.26%	20	**3.33**	297	1.67
Total	121181	8285	6.84%	22579	**2.73**	263687	2.34

Column “Ratio” denotes the ratio of the number of higher-order functions dividing the number of all functions. #Total calls and #Avg. calls denote the number of all function calls and the average number of calls per definition. *HOFs* and *FOFs* stand for higher-order functions and first-order functions, respectively

As shown in Table 2, the total number of calls of higher-order functions is lower than that of first-order functions. Comparing the average number of calls per definition, we find that the average calls of higher-order functions and first-order functions are 2.73 and 2.34, respectively. Among 35 projects, the average calls per higher-order function are higher than the average calls per first-order function. The average calls suggest that higher-order functions are called more frequently than first-order functions.

Higher-order functions exist in all 35 projects in the study: the ratio of definitions of higher-order functions ranges from 1.50% to 24.82%. On average, 6.84% of functions are defined as higher-order functions. The average calls per function show that higher-order functions are called more frequently than first-order functions.

3.2 RQ2. How are Higher-Order Functions Defined?

Method We explore the types of defining higher-order functions via answering RQ2. 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 II`, a function definition returns at least one function with no function input;
- `Type III`, a function definition takes at least one function as a parameter and returns functions.

Result and analysis Figure 2 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. Figure 3 presents the percentage of function definitions of higher-order functions in all projects by categorizing the definition types.

```
/* 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. 2 Examples of higher-order functions in Type I, Type II, and Type III

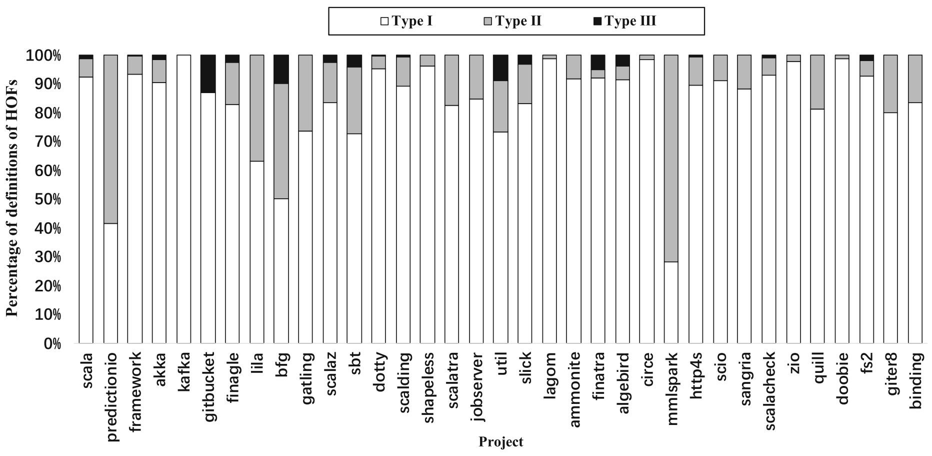


Fig. 3 Percentage of function definitions of higher-order functions in all projects by categorizing the definition types

Among 35 projects under consideration, we can observe that 27 projects contain over 80% of definitions in `Type I` while 13 projects contain 90% of definitions in `Type I`. In all 35 projects, 87.35% of definitions of higher-order functions belong to `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 percentage of `Type I` of higher-order functions is the highest and reaches 100%; Meanwhile, in 17 projects, over 10% of definitions of higher-order functions belong to `Type II` and in 6 projects, over 20% of definitions of higher-order functions belong to `Type II`. In project `mmlspark`, the percentage of `Type II` of higher-order functions is the highest and reaches 71.67%; in addition, `Type III` of higher-order functions account for over 10% in one project.

We further show the percentage of calls of higher-order functions in three types in Fig. 4. We can find that 25 out of 35 projects contain over 80% of calls of higher-order functions in `Type I` while 21 projects contain 90% of calls in `Type I`. In four projects, `kafka`, `finatra`, `circe`, and `binding`, the percentage of `Type I` of higher-order functions is the highest and reaches 100%. In all 35 projects, 90.66% of calls of higher-order functions belong to `Type I`. Meanwhile, in 14 projects, over 10% of calls of higher-order functions belong to `Type II`, and in 8 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 Figs. 3 and 4, `Type I` accounts for the highest percentage among all the definitions and calls of higher-order functions.

Among all higher-order functions in the study, only taking functions as parameters is the most common type of defining higher-order functions (87.35% of definitions with 90.66% of calls). A higher-order function that takes at least one function as a parameter and returns functions is not frequently defined and called.

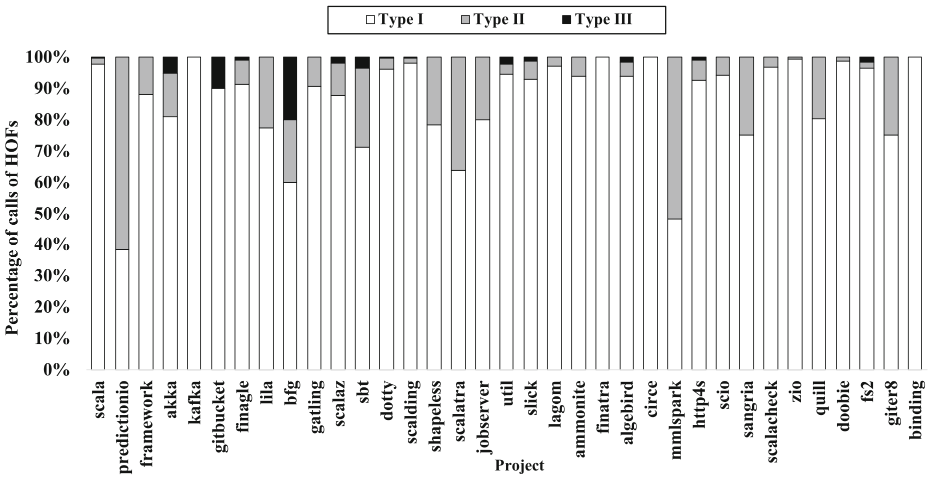


Fig. 4 Percentage of function calls of higher-order functions in all projects by categorizing the definition types

3.3 RQ3. How do Definitions of Higher-Order Functions Distribute?

Method We characterize the definitions of higher-order functions via three measurements, the LoC, the Cyclomatic complexity, and the warnings in the code style. These measurements are directly extracted from the source code of Scala functions (in Section 2.1). For each measurement, we compare the percentage between higher-order functions and first-order functions.

Result and analysis To compare these measurements between higher-order functions and first-order functions, we present minimum, median, maximum, average, and standard deviation values in Table 3. We conducted the Wilcoxon rank-sum test between higher-order functions and first-order functions and the p-values in Table 3. As shown in Table 2, the numbers of higher-order functions and first-order functions are different. Thus, we chose the Wilcoxon rank-sum test, which is a non-parametric and non-paired test (Wilcoxon 1992).

The p-values in the Wilcoxon rank-sum test in Table 3 show that the measurements of LoC, complexity, and warnings between higher-order functions and first-order functions

Table 3 The p-value between higher-order functions and first-order functions of LoC, complexity, and warnings in the code style as well as their minimum, median, maximum, average, and standard deviation ($Std.$) values

Measurement	Function	Min	Median	Max	Average	Std.	p-value
LoC	Higher-order	1	2	**144**	**5.14**	**7.75**	2.34E-21
	First-order	1	2	781	5.70	12.62	
Complexity	Higher-order	1	1	**17**	**1.35**	**1.02**	2.03E-24
	First-order	1	1	132	1.56	2.02	
#StyleWarnings	Higher-order	0	0	**29**	**0.38**	**1.06**	2.09E-58
	First-order	0	0	93	0.54	1.31	

are significantly different. We observed that minimum and median values between higher-order functions and first-order functions are the same. The reason for this observation is that most of the functions are short, simple, and non-risky. From the median values of warnings in the code style, half of the higher-order functions and first-order functions contain no issues in code style. The average and the standard deviation values show that the measurements of higher-order functions, including LoC, complexity, warnings, are lower than those of first-order functions. We showed illustrations on three measurements as follows.

1) Executable Lines of Code In Table 3, the medians of the LoC, the Cyclomatic complexity, and the warnings in the code style are 2, 1, and 0, respectively. To illustrate the measurements, we show the percentage of values that are over the medians.

Figure 5 presents the accumulative percentage of the definitions of higher-order functions that are no less than three lines. In 10 out of 35 projects, the percentage of functions with three or more lines in higher-order functions is lower than that in first-order functions. This shows that in these 25 projects, higher-order functions are longer than first-order functions in terms of functions over the medians. In 21 projects, the percentage of functions with 20 or more lines in higher-order functions is lower than that in first-order functions. In 22 projects, the percentage of functions with over 30 lines in higher-order functions is lower than that in first-order functions.

As shown in Fig. 5, there are over 10% of definitions of higher-order functions with over 10 lines in 28 out of 35 projects; in 29 projects, definitions of first-order functions with over 10 lines are over 10%. In one project, `predictionio`, there are over 70% of definitions of higher-order functions with over 10 lines. Considering the number of lines, in 9 out of 35 projects, over 10% of definitions of higher-order functions contain over 20 lines; in 6 projects, over 10% contain over 30 lines. In 10 out of 35 projects, i.e., `framework`, `scalaz`, `util`, `slick`, `bfg`, `shapeless`, `scalatra`, `finatra`, `circe`, and `giter8`, there exists no definition of higher-order functions with over 30 lines. For first-order functions, there are over 10% of definitions of first-order functions with over 20 lines in 4 out of 35 projects; there is no project contains over 30 lines over 10%.

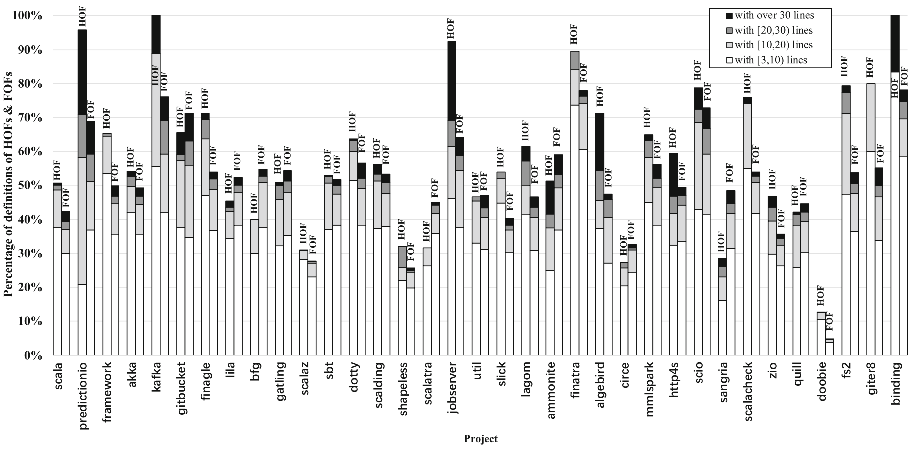


Fig. 5 Accumulative percentage of function definitions for higher-order functions and first-order functions by counting LoC of no less than three lines

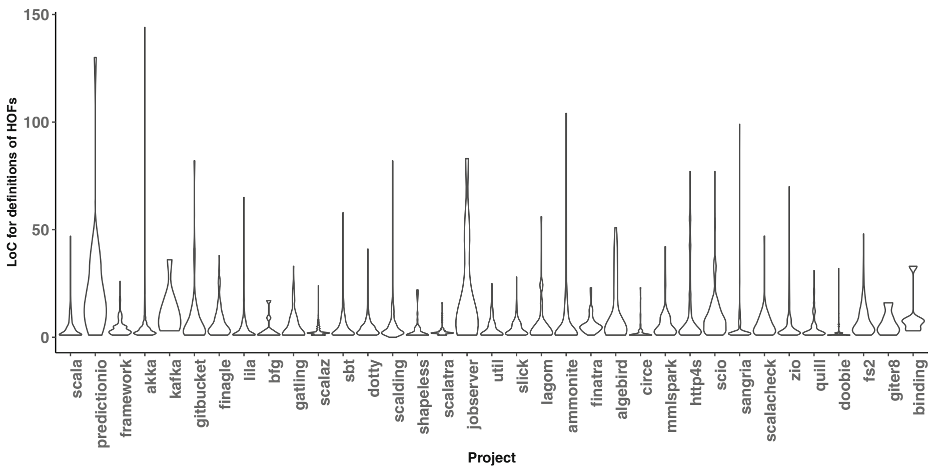


Fig. 6 Violin-plots of LoC for function definitions of higher-order functions. The width of each bar is equal, which denotes the maximum number of definitions with the same LoC inside one project

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. 6. In each violin-plot, the LoC value with the broadest line indicates the LoC that appears for the most times. Among 35 projects, 28 projects show a similar shape, where the data are mainly concentrated at the bottom. The highest LoC reaches 144 in Project `akka`. 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 paths. Figure 7 presents the accumulative percentage of definitions of higher-order functions in Cyclomatic complexity. In 19 out of 35 projects, the

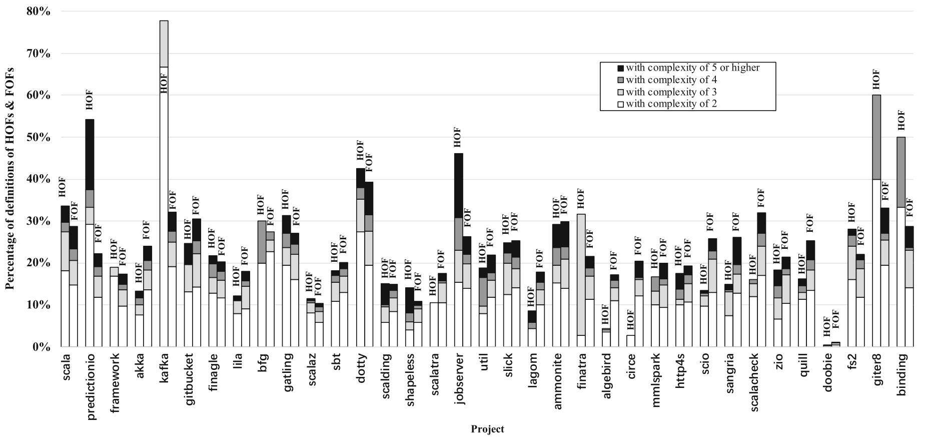


Fig. 7 Accumulative percentage of function definitions for higher-order functions and first-order functions by counting Cyclomatic complexity of no less than two

percentage of functions with the complexity of two or more in higher-order functions is lower than that in first-order functions. In 31 out of 35 projects, there are over 10% of definitions of higher-order functions with the complexity of two or more; in 15 projects and 9 projects, there are over 20% and 30% of definitions of higher-order functions with the complexity of two or more, respectively. As for first-order functions, there are over 10% of definitions of first-order functions with the complexity of two or more in 34 out of 35; in 24 projects and 5 projects, there are over 20% and 30% of definitions of first-order functions with the complexity of two or more, respectively. As shown in Fig. 7, 16 out of 35 projects have over 10% of definitions of higher-order functions with the complexity of three or more; four projects have over 10% of definitions of higher-order functions with the complexity of four or more; and two projects have over 10% of definitions of higher-order functions with the complexity of five or more. As for first-order functions, 21 projects have over 10% of definitions with the complexity of three or more; 1 project (Project `dotty`) has over 10% of definitions with the complexity of four or more; and no project has over 10% of definitions with the complexity of five or more.

Through analysis of the complexity of higher-order functions, we find that most higher-order functions favor low complexity. We note that a higher-order function can actually represent a group of first-order functions (Lincke and Schupp 2012); the linear increment of the complexity in a higher-order function may represent an exponential increment of the complexity in a first-order function.

Figure 8 presents the violin-plots of Cyclomatic complexity for the definitions of higher-order functions. From the figure, 28 projects have a similar distribution, where the data are mainly concentrated at the bottom. Most of the higher-order functions tend to be simple code structure with the Cyclomatic complexity of less than five. Six projects behave differently: plots of Projects `kafka`, `jobserver`, `finatra`, `giter8`, and `binding` aggregate in the middle or the top, not the bottom. The highest complexity reaches 17 in Project `quill`.

3) Warnings in the Code Style The code style is also used to measure the quality of the source code. Bacchelli and Bird (2013a) and Gousios et al. (2016) have shown that code style issues can reveal potential risks in the source code and may affect the code review and

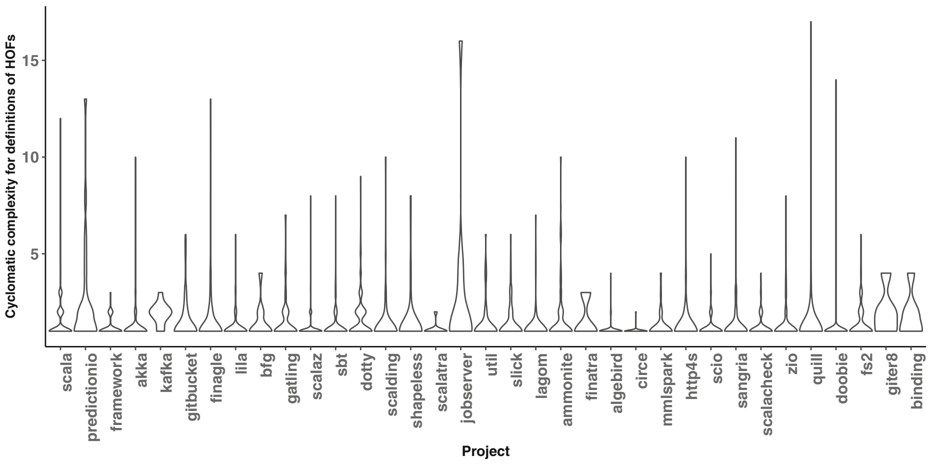


Fig. 8 Violin-plots of Cyclomatic complexity for the definitions of higher-order functions. The width of each bar is equal, which denotes the maximum number of definitions with the same complexity inside one project

code integration. Zou et al. (2019) 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 in the code style to measure potential risks of source code.

Figure 9 presents the accumulative percentage of definitions of higher-order functions by counting the warnings in the code style. In 23 out of 35 projects, there are fewer warnings in the code style in higher-order functions than in first-order functions. In 34 projects, over 10% of definitions of first-order functions contain one or more warnings in the code style; over 20% and 30% of definitions of first-order functions in 25 projects and 15 projects, respectively, contain one or more warnings in the code style. As shown in Fig. 9, there are over 10% of definitions of higher-order functions with the warnings of two or more in 12 out of 35 projects; in three projects, `kafka`, `ammonite`, and `giter8`, over 10% of definitions of higher-order functions contain over three warnings in the code style. In one project, `kafka`, over 65% of definitions contain one or more warnings in the code style. This observation reveals that code style issues widely exist in most of the higher-order functions in Project `kafka`. In Project `binding`, these is no warning in the code style since there are only 6 higher-order functions. For first-order functions, there are over 10% of definitions with the warnings of two or more in 12 projects; in one project, Project `sangria`, over 10% of definitions contain over three warnings in the code style.

Figure 10 presents the violin-plots of warnings in the code style for the definitions of higher-order functions. Among 35 projects, 27 projects have a similar distribution, where the data are mainly concentrated at the bottom. That observation of most of the higher-order functions with the warnings of zero shows that these higher-order functions have no code style issue. The highest number of warnings in the code style reaches 29 in Project `dotty`; that is, a higher-order function contains 29 reported warnings in the code style. A warning in the code style in a function reveals that there is a potential risk to the code quality in the future (McIntosh et al. 2016; Bacchelli and Bird 2013b; Rigby and Storey 2011). There are warnings in over 10% of higher-order functions in 29 projects, and over 10% of first-order functions in 34 projects. These higher-order functions and first-order functions should be cautiously maintained in daily development.

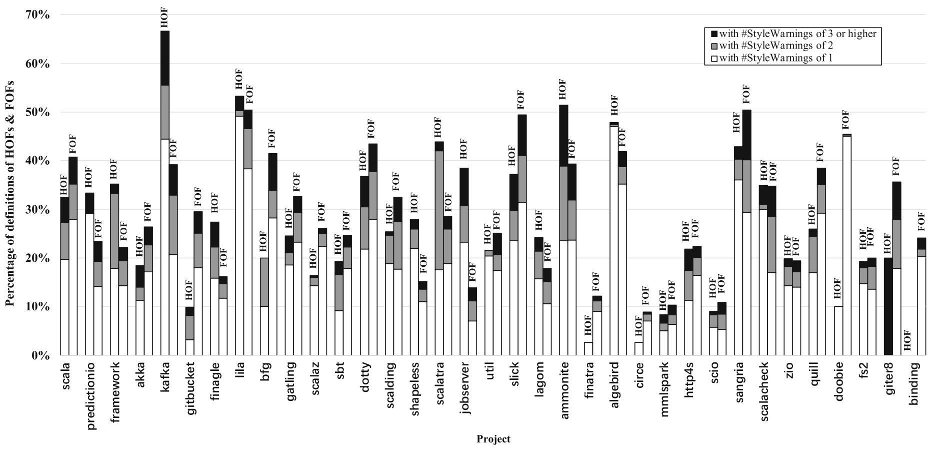


Fig. 9 Accumulative percentage of function definitions for higher-order functions and first-order functions by counting the warnings in the code style of no less than one

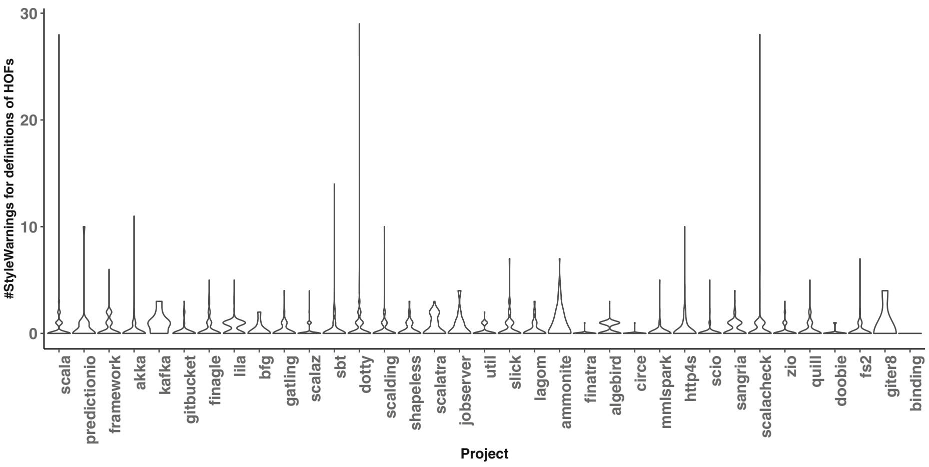


Fig. 10 Violin-plots of warnings in the code style for the definitions of higher-order functions. The width of each bar is equal, which denotes the maximum number of definitions with the same number of warnings in the code style inside one project.

The average and the standard deviation values show that the measurements of higher-order functions, including LoC, complexity, warnings, are lower than those of first-order functions. For functions whose measurements are the medians, higher-order functions are lower in LoC, complexity, and warnings in code style than first-order functions in 10, 19, and 23 projects, respectively.

3.4 RQ4. Which Factor Correlates with the Calls of Higher-Order Functions?

Method We intended to find out the factors that correlate with the number of calls of higher-order functions. Firstly, we build a multivariate linear regression model to understand the correlation between the number of calls with three measurements, i.e., LoC, Cyclomatic complexity, and #StyleWarnings. Secondly, 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. Thirdly, we illustrate the number of function calls per definition with violin-plots.

Result and Analysis

1) Multivariate Linear Regression Analysis on All Projects We used the multivariate linear regression model to measure the correlation between the number of function calls and three measurements that potentially affect the calling. Multivariate linear regression model is a regression model that can estimate linear correlations between one or more dependent variables and multiple independent variables (Cohen et al. 2013). The correlation coefficient for each independent variable is estimated by considering all variables.

In our model, the calls of higher-order functions are considered as a dependent variable while the three measurements are considered as independent variables. We added another two variables as confounding factors, i.e., the number of authors per function and the number of commits per function. We chose these two confounding factors since the counts of authors or commits can directly link to functions and may correlate with the calls of higher-order functions. A *confounding factor* in a regression model is a variable that influences both dependent variables and independent variables; confounding is a causal concept that may cause a spurious correlation and cannot be described as acorrelation (Cohen et al. 2013).

Table 4 presents the multivariate linear regression model between the number of calls and the three measurements for each higher-order function. We built two models with or without confounding factors, respectively. In the model with confounding factors, the correlation between the number of functions calls and the complexity for each higher-order function is 0.7323. The *p*-value 0.0038 shows the correlation is statistically significant and the number of calls for one higher-order function highly correlates with the complexity. For the other two measurements, LoC and #StyleWarnings, the *p*-values are 0.4988 and 0.3664; that is, there is no statistically significance.

For three measurements, changes on coefficient between two models with or without confounding factors are 21.77%, 4.41%, and 8.11%, respectively. For regression models, a change over 10% suggests that the independent variable may involve a spurious correlation due to confounding factors (Budtz-Jorgensen et al. 2007; Lee 2015). That is, the correlation between the number of calls and LoC may be caused by the two confounding factors.

2) Correlation Analysis on Each Project To further understand the number of calls in each project, we use the Spearman's rank correlation coefficient to quantify the correlation between the number of calls and each measurement that potentially correlates with the calls. Spearman's rank correlation coefficient is a non-parametric measure of the statistical correlation between ranks of two variables (Walpole et al. 2007). The correlation coefficient, varying from -1 to 1, is calculated with the covariance of ranks of two given variables. The absolute value of the coefficient indicates the degree of correlation between two variables: zero means no correlation and one means completely correlated. 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. We consider a *p*-value less than 0.05 as statistically significant.

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

Table 4 Multivariate linear regression model between the number of calls and the three measurements (LoC, complexity, and #StyleWarnings) for each higher-order function

Variable	Model *with* confounding factors		Model *without* confounding factors		Change
	Coefficient	*p*-value	Coefficient	*p*-value	
(Intercept)	1.4090	0.0003	2.2390	0.0000	–
LoC	−0.0226	0.4988	−0.0177	0.5971	21.77%
Complexity	0.7323	0.0038	0.7000	0.0058	4.41%
#StyleWarnings	−0.1992	0.3664	-0.2153	0.3296	8.11%
#Authors	0.4934	0.0000	–	–	–
#Commits	−0.0513	0.0270	–	–	–

Two models are built, with or without confounding factors. The two confounding factors are used, #Authors and #Commits. *(Intercept)* denotes the intercept of the linear model and *change* denotes the percent of the coefficient change with or without confounding factors

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. We labeled a coefficient in bold if its *p*-value is less than 0.05

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
predictionio	0.2747	0.1938	**0.4075**	0.0481	−0.1263	0.5566
framework	0.0515	0.4737	**−0.1757**	0.0138	0.0229	0.7502
akka	**0.1455**	0.0009	0.1019	0.0203	−0.0028	0.9490
kafka	0.6610	0.0526	0.2219	0.5661	0.2155	0.5776
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
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
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
finatra	0.1702	0.3069	−0.0168	0.9202	−0.0972	0.5617
algebird	**−0.1711**	0.0093	**0.1603**	0.0150	**−0.2263**	0.0005
circe	**0.6758**	<2.20E-16	0.1816	0.0543	−0.0873	0.3579
mmlspark	0.2067	0.1131	0.0024	0.9858	−0.0450	0.7330
http4s	**0.3073**	7.72E-05	**0.1734**	0.0283	**0.2101**	0.0077
scio	−0.0317	0.6947	**0.1629**	0.0421	−0.0784	0.3309
sangria	0.0758	0.3392	0.0736	0.3535	**−0.2894**	0.0002
scalacheck	0.1045	0.3008	0.1759	0.0801	−0.1090	0.2804
zio	**0.2683**	4.10E-12	**0.2191**	1.83E-08	0.0524	0.1836
quill	**−0.2357**	0.0087	−0.1026	0.2587	**−0.1868**	0.0386
doobie	**−0.2052**	1.97E-05	**0.1182**	0.0146	0.0792	0.1028
fs2	0.1487	0.0694	0.0659	0.4228	0.0486	0.5545
giter8	0.5407	0.3467	0.1111	0.8588	0.7454	0.1482
binding	0.0924	0.8618	0.0984	0.8529	n/a †	n/a †

†This value is not available since the number of warnings in the code style is zero

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 show the statistical significance: the number of calls of higher-order functions correlates with the LoC in 19 out of 35 projects, with the Cyclomatic complexity in 15 projects, and with #StyleWarnings in 12 projects. For the LoC, coefficients of 15 out of 19 projects with statistical significance have positive correlations of over 0.17. For the Cyclomatic complexity, coefficients of 14 out of 15 projects with statistical significance have positive correlations. For #StyleWarnings, coefficients of 5 out of 12 projects with statistical significance show positive correlations with the number of function calls while coefficients of the other 7 projects show negative correlations.

3) Illustration on Calls per Function Definition We illustrated the distribution of numbers of function calls for different values of the three measurements. Figure 11 presents the illustration of numbers of function calls for different values of LoC. The width in each violin represents the frequency of the corresponding number of function calls. As shown in Fig. 11, the shape of the violin gradually narrows when the LoC increases. Most of the higher-order functions are called for once or twice. When the LoC is over 25, each violin basically behaves like a straight line; that is, the destiny is low. We 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.

Figure 12 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. 12, there exists a higher-order function with Cyclomatic complexity of four that has the most calls over 1,000. 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.

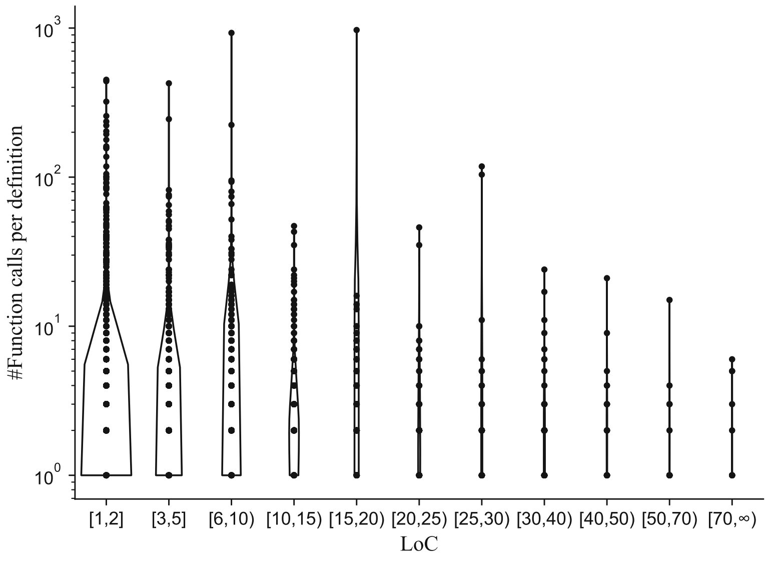


Fig. 11 Number of function calls per definition for LoC

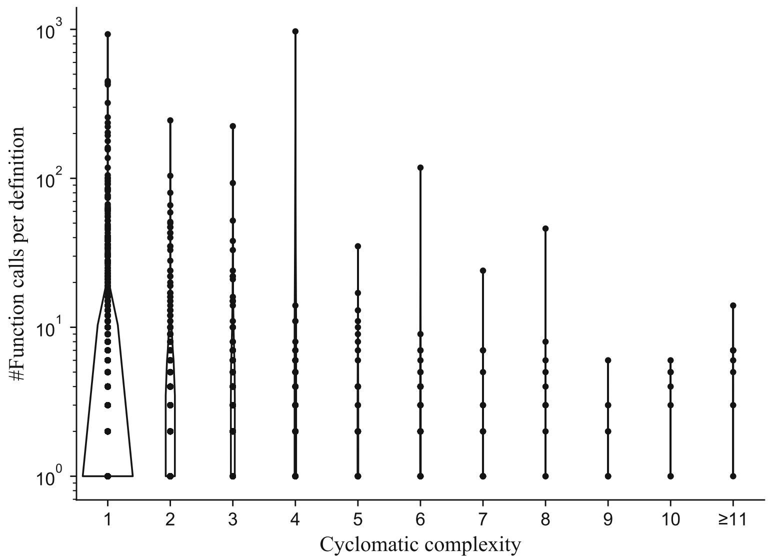


Fig. 12 Number of function calls per definition for Cyclomatic complexity

Figure 13 presents the illustration of the numbers of function calls with #StyleWarnings. As shown in Fig. 13, most higher-order functions have no code style warning. Higher-order functions with warnings concentrated on the number of one or two. The majority of the higher-order functions with warnings have one or two warnings. Meanwhile, as #StyleWarnings increases, the maximum value of higher-order function calls fluctuates. One possible

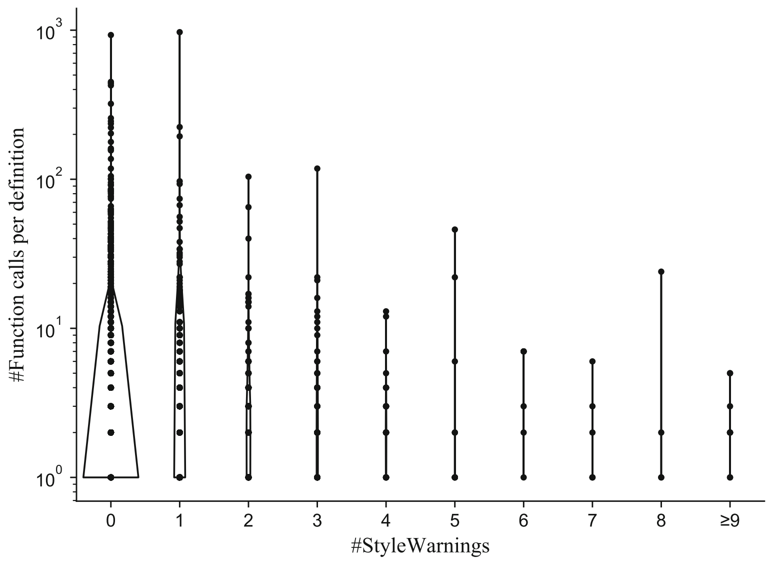


Fig. 13 Number of function calls per definition for #StyleWarnings

reason for this fact is that the requirements of the code style are not consistent among all projects in the study.

The regression analysis on all projects suggests that the number of calls for a higher-order function highly correlates with the Cyclomatic complexity. Results on individual projects show that the correlations with the number of executable lines of code are positive in most projects; all correlations but one with the Cyclomatic complexity are positive; the number of warnings in the code style contains both positive correlations and negative correlations.

3.5 RQ5. How do Developers Contribute to Defining and Calling Higher-Order Functions?

Method We extracted and counted the number of function definitions and calls that are contributed by each developer. Firstly, we divide functions in three categories to identify developer contributions. Secondly, we compare definitions and calls of higher-order functions and first-order functions that are made by top 20% developers. Thirdly, we illustrate the definitions and calls per developer in each project.

Result and analysis Among a large number of function calls, *how are function definitions and calls contributed by developers*? As mentioned in Section 2.1, we collected developer information via the Git API. We identified the developer, who has written the most changes to a definition or a call of a higher-order function, as the author of the definition or the call of the higher-order function.

1) Functions that are Called by Different Developers Figure 14 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 by other developers except the author of the definition, and a definition that is called by both its author and other developers. In 24 projects, the percent of functions that are called by both authors and other developers in higher-order

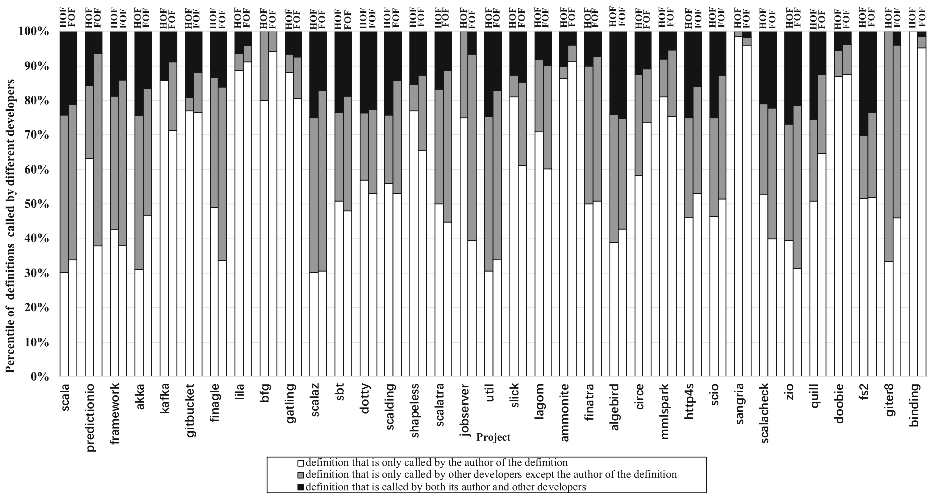


Fig. 14 Percentage of function definitions that are called by different types of developers

functions is higher than that in first-order functions. We observed that there are 22 projects, whose over 50% definitions of higher-order functions are only called by authors of these definitions; there are 20 projects, whose over 50% definitions of first-order functions are only called by authors. In Project `binding`, the percentage of the first category of higher-order functions is the highest and reaches 100%. This observation shows that the definitions of higher-order functions are more frequently called by the author of these definitions than by other developers. In addition, in 20 projects, over 20% of definitions of higher-order functions belong to the second category and in 14 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 6 lists the numbers of function calls based on three types of definitions and the overlap of developers. Among 105 mini-bar charts, 26 charts are blank since there is no such function in the corresponding category, including 5 charts in `Type II` and 21 in `Type III`. Among the 79 non-blank mini-bar charts, 39 charts show that most of the calls are made by the authors of functions definitions; 6 charts shows that most of calls are made by other developers; and 34 charts shows that most of the calls are made by both the authors of definitions and other developers. This fact indicates that most of the calls of higher-order functions are made by the same developers who have written the higher-order functions.

As shown in Table 6, there also exist 5,236 function calls of higher-order functions that are made by developers other than the authors of definitions, including 4841 calls of `Type I`, 336 calls of `Type II`, and 59 calls of `Type III`. This observation indicates that developers have maintained the collaboration between defining and calling functions. We can also observe that among all the charts in `Type I`, the higher-order functions in 14 out of 35 projects are mostly called by their authors; in `Type II`, the higher-order functions in 20 projects are mostly called by their authors. However, in the total data of 21 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 five large projects, including `scala`, `scalaz`, `sbt`, `zio`, and `doobie` are mostly called by both authors and other developers. This indicates that in several large projects, such as Project `scala`, developers may tend to work more collaboratively than in small projects.

2) Definitions and Calls by Top 20% Developers 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 incomes are made from 20% of products (Koch 2011; Reed 2001).

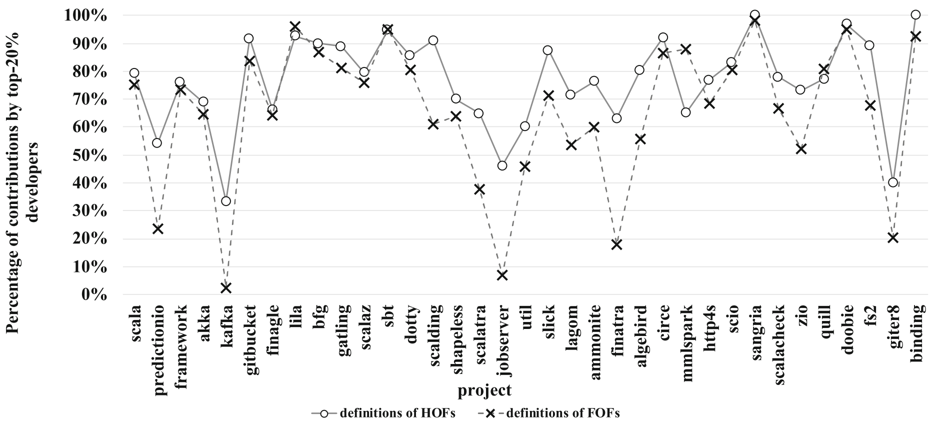
Figure 15 shows the percentage of definitions of higher-order functions and first-order functions by top 20% developers, who have defined higher-order functions (Fig. 15(a) and first-order functions (Fig. 15b). The y-axis in Fig. 15 is the ratio between the number of contributed higher-order (or first-order) functions and the number of total higher-order (or first-order) functions. That is, we focused on how many higher-order (or first-order) functions are contributed by top 20% developers. From Fig. 15a, we found that in 32 out of 35 projects (91.43%), the top 20% developers who have contributed the most higher-order functions have defined more percent of higher-order functions than first-order functions. From Fig. 15b, in 20 projects (57.14%), the top 20% developers who have contributed the most first-order functions have defined more percent of first-order functions than higher-order functions.

Table 6 Function calls of higher-order functions based on the types of `Type I`, `Type II`, and `Type III`

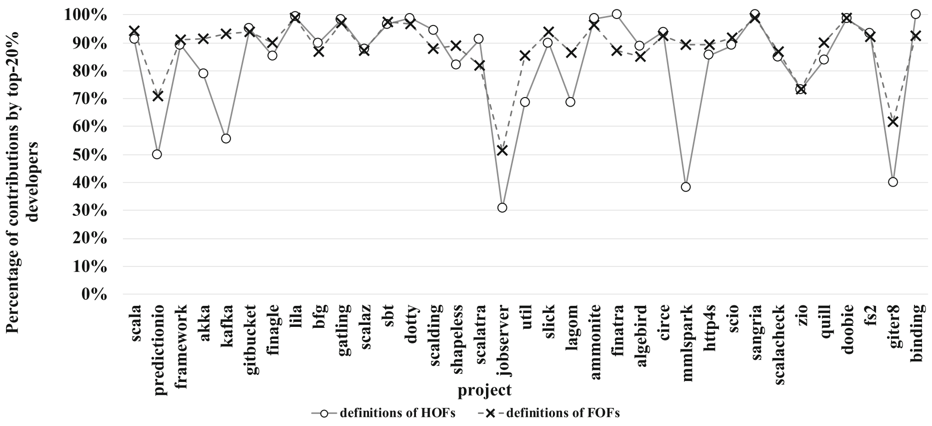
Index	Project	#Calls	#Calls of Type I					#Calls of Type II					#Calls of Type III				
			All	Self	Others	Both		All	Self	Others	Both		All	Self	Others	Both	
1	scala	5162	5047	316	2127	2604		103	34	41	28		12	0	12	0	
2	predictionio	52	20	4	0	16		32	22	10	0		0	0	0	0	
3	framework	266	234	64	63	107		32	9	7	16		0	0	0	0	
4	akka	1292	1047	159	312	576		180	3	35	142		65	7	2	56	
5	kafka	25	25	16	0	9		0	0	0	0		0	0	0	0	
6	gitbucket	543	489	218	11	260		0	0	0	0		54	34	0	20	
7	finagle	204	186	82	60	44		16	13	3	0		2	2	0	0	
8	lila	505	391	295	7	89		114	111	3	0		0	0	0	0	
9	bfg	5	3	2	1	0		1	1	0	0		1	1	0	0	
10	gatling	345	313	189	6	118		32	32	0	0		0	0	0	0	
11	scalaz	4191	3677	364	885	2428		438	36	102	300		76	4	29	43	
12	sbt	1989	1415	476	280	659		505	98	84	323		69	24	6	39	
13	dotty	770	740	266	101	373		28	12	12	4		2	2	0	0	
14	scalding	846	829	209	154	466		13	11	2	0		4	0	0	4	
15	shapeless	46	36	28	0	8		10	4	6	0		0	0	0	0	
16	scalatra	33	21	14	5	2		12	1	2	9		0	0	0	0	
17	jobserver	30	24	16	8	0		6	4	2	0		0	0	0	0	
18	util	500	473	37	101	335		15	4	8	3		12	2	10	0	
19	slick	391	363	168	7	188		23	15	1	7		5	5	0	0	
20	lagom	35	34	20	5	9		1	1	0	0		0	0	0	0	
21	ammonite	133	125	42	1	82		8	8	0	0		0	0	0	0	
22	finatra	14	14	6	6	2		0	0	0	0		0	0	0	0	
23	algebird	176	165	41	36	88		8	8	0	0		3	1	0	2	
24	circe	68	68	25	10	33		0	0	0	0		0	0	0	0	
25	mmlspark	87	42	22	1	19		45	26	5	14		0	0	0	0	
26	http4s	273	253	69	55	129		17	10	3	4		3	3	0	0	
27	scio	435	410	94	61	255		25	20	0	5		0	0	0	0	
28	sangria	133	100	99	1	0		33	33	0	0		0	0	0	0	
29	scalacheck	239	231	62	61	108		8	3	1	4		0	0	0	0	
30	zio	1524	1512	204	319	989		12	6	6	0		0	0	0	0	
31	quill	213	171	51	54	66		42	21	0	21		0	0	0	0	
32	doobie	1485	1464	417	45	1002		21	9	0	12		0	0	0	0	
33	fs2	545	526	99	57	370		10	4	2	4		9	4	0	5	
34	giter8	4	3	2	1	0		1	0	1	0		0	0	0	0	
35	binding	20	20	20	0	0		0	0	0	0		0	0	0	0	
Total		22579	19439	4196	4841	11434		1791	559	336	896		317	89	59	169	

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 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 by both authors and other developers

Similar to the definition, we used Fig. 16 to illustrate the percentage of calls of higher-order functions and first-order functions by top 20% developers, who have called higher-order functions (Fig. 16a) and first-order functions (Fig. 16b). From Fig. 16a, we found



(a) By top 20% developers who have defined **higher-order functions**



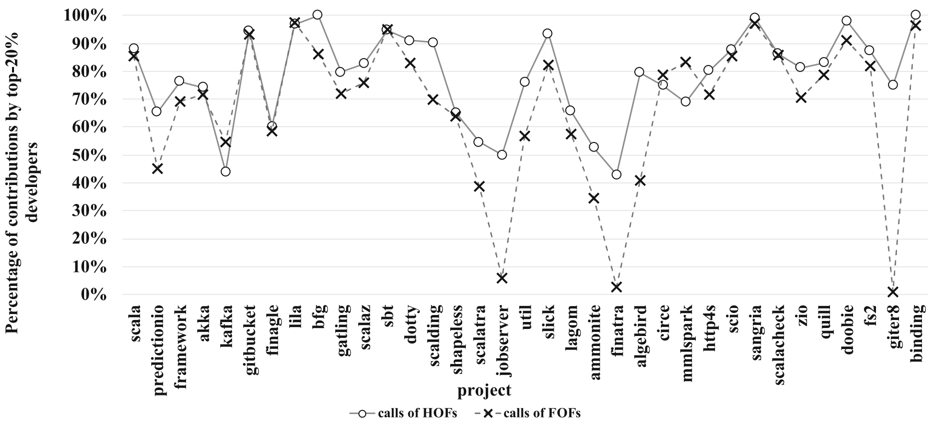
(b) By top 20% developers who have defined **first-order functions**

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

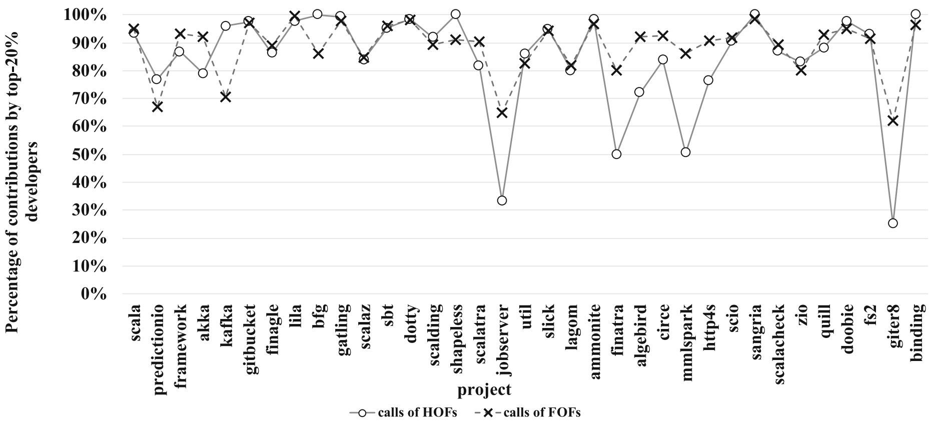
that in 31 out of 35 projects (88.57%), the top 20% developers who have contributed the most higher-order functions have called more percent of higher-order functions than first-order functions. From Fig. 16b, in 19 projects (54.28%), the top 20% developers who have contributed the most first-order functions have called more percent of first-order functions than higher-order functions.

Figures 15 and 16 show that top 20% authors of higher-order functions have defined and called more percent of higher-order functions than first-order functions in 32 and 31 projects, respectively; top 20% authors of first-order functions have defined and called more percent of first-order functions than higher-order functions in 20 and 19 projects. This reveals that authors of higher-order functions favor using higher-order functions, comparing with authors of first-order functions.

3) Definitions and Calls per Developer To understand the contribution of function definitions, we counted how many functions are defined by each developer. Figure 17 presents the box-plots of numbers of definitions by each developer on the log scale. In 29 projects,



(a) By top 20% developers who have called **higher-order functions**



(a) By top 20% developers who have called **first-order functions**

Fig. 16 Percentage of calls of higher-order functions and first-order functions by top 20% developers

the median number of definitions of higher-order functions per developer is lower than that of first-order functions. As shown in Fig. 17, the median number of definitions of higher-order functions by each developer is no more than 20 in 34 out of 35 projects; one exception is Project `sangria`, whose median is 80.5. In 14 projects, 25% of developers have contributed over 10 definitions of higher-order functions; in 27 projects, 25% of developers have contributed over 10 definitions of first-order functions. We can observe that several developers have indeed defined many higher-order functions in their daily development.

We also illustrated the contribution of calls of higher-order functions and first-order functions. Figure 18 presents the box-plots of numbers of function calls by each developer on the log scale. In 30 projects, the median number of calls of higher-order functions per developer is lower than that of first-order functions. In 16 projects, 25% of developers have contributed over 10 calls of higher-order functions; in 31 projects, 25% of developers have contributed over 10 calls of first-order functions.

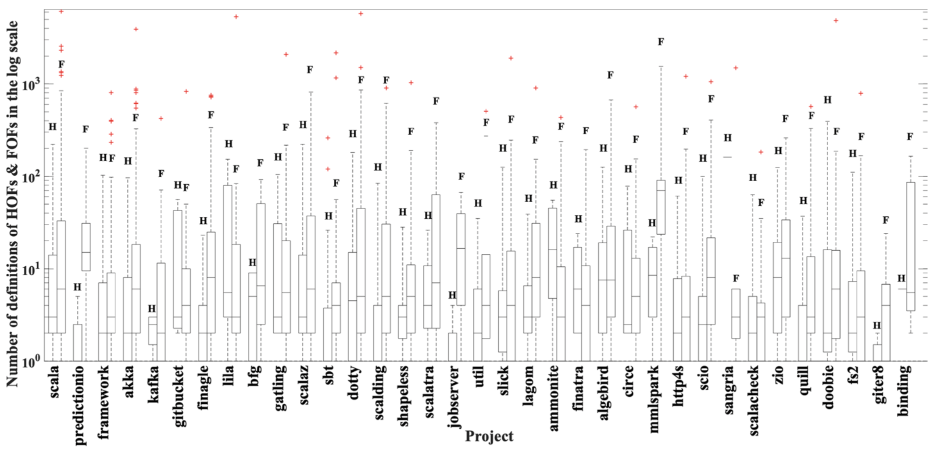


Fig. 17 Box-plots of function definitions of per developer on the log scale

Among all calls of higher-order functions, 43.82% 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. Results suggest that top 20% authors of higher-order functions have defined and called more percent of higher-order functions than first-order functions in 32 and 31 projects, respectively.

3.6 Discussion

Using higher-order functions is not easy (Lincke and Schupp 2012). In Table 7, we summarized the results and brief implications of higher-order functions according to RQs in Sections 3.1–3.5.

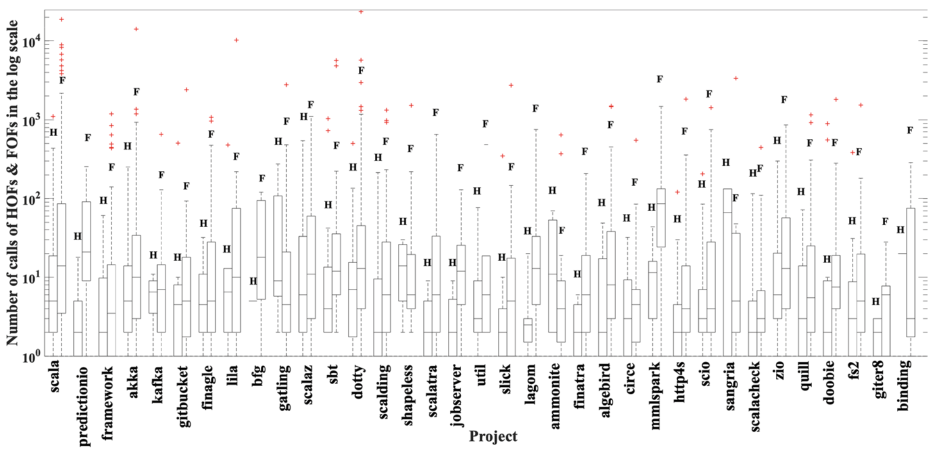


Fig. 18 Box-plots of function calls per developer on the log scale

Table 7 Brief implications based on the exploratory study with five research questions

RQ	Implication
RQ1. How many higher-order functions are there in Scala projects?	In all 35 Scala projects, the ratio of definitions of higher-order functions ranges from 1.50% to 24.82%. This indicates that as a language feature of Scala, higher-order functions are accepted and used in all projects in our study.
	The average number of calls per function suggests that higher-order functions (2.73 calls per definition) are called more frequently than first-order functions(2.34 calls per definition). Using higher-order functions can help if code reuse and maintenance is crucial to the project (Richardson 2017).
RQ2. How are higher-order functions defined?	Functions that only take functions as parameters are the most common type of definitions among all higher-order functions in the study; functions that both take functions as a parameter and return functions are not frequently defined and called. To assist developers who need all types of higher-order functions, software companies and organizations can provide training to improve the skills of mastering higher-order functions (if necessary).
RQ3. How do definitions of higher-order functions distribute?	This study shows that the average and the standard deviation values of three measurements (including LoC, complexity, warnings in code style) in higher-order functions are lower than those of first-order functions. This suggests higher-order functions are not longer, more complex, or riskier than first-order functions. Meanwhile, this partially shows the evidence that higher-order functions may be used more in the future.
RQ4. Which factor correlates with the calls of higher-order functions?	Regression analysis on all projects shows that the number of calls for a higher-order function highly correlates with the Cyclomatic complexity. This suggests a complex higher-order function does not hinder its use.
	Correlation analysis on individual projects shows that definitions with high LoC or high Cyclomatic complexity correlate with the callings of these functions. When a developer defines a higher-order function, the definition could be long or complex if it is necessary.
RQ5. How do developers contribute to defining and calling higher-order functions?	Among all calls of higher-order functions, 43.82% calls are made by the same developers who have defined the functions. Top 20% developers of higher-order functions have defined or called more percent of higher-order functions than first-order functions. This suggests that higher-order functions are not used by every developer. The community of programming languages with the feature of higher-order functions like Scala may promote this language feature to attract new users.

Besides the results in Table 7, we listed our suggestions on using Scala in practice as follows.

- **Using complex higher-order functions if necessary**. Our study analyzed the correlation between the number of function calls and the Cyclomatic complexity of definitions. Results in RQ4 shows complex higher-order functions highly correlate with the function calls. Therefore, developers do not need to reject using complex higher-order functions if these functions are necessary in development.
- **Building collaboration among developers**. This study examined the developer contributions of higher-order functions in Scala projects. RQ5 reveals that 43.82% of higher-order functions are called by the same developers who have made their definitions; authors of higher-order functions favor using higher-order functions than first-order functions. We suggest that developers continue building collaborations among developers and promote their code to enlarge the user group of their higher-order functions.
- **Providing training for language features**. In this study, we showed the existence of higher-order functions in 35 Scala projects via investigation on definitions and calls. Considering the current use of higher-order functions in Scala programs, software companies or organizations can provide training of language features, such as higher-order functions in Scala. This can bring in new developers to learn and apply higher-order functions to their projects.

4 Threats to Validity

We conducted an exploratory study on using higher-order functions in Scala programs. We discussed the threats to the validity of our work in three aspects.

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 techniques or tools that can be used as measures, such as the Halstead complexity to measure the functional complexity based on parsing operators (Albrecht and Gaffney 1983). We plan to involve other measures in further work. Meanwhile, the warning in the code style reveals a potential risk to quality. Our study has not validated such risks due to the unavailable data of quality in the future. In Section 2.1, 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 affect the computation of developer contributions, e.g., the result in Section 3.4. In developer extraction, we parsed all logs in the project history. However, several commits may be deleted by the project manager. In this case, developers in the deleted logs cannot be extracted. We followed existing works (Zhao et al. 2017; Zou et al. 2019; Fry et al. 2020) to skip developers in the deleted logs.

Threats to internal validity The correlation analysis in the paper may be biased due to 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, such as the number of calls and the LoC. The experimental result in the paper can be viewed as an observation on the linear correlation and the impact of confounding variables could be further explored.

Threats to external validity Our study selected 35 Scala projects according to the number of stars from GitHub. Therefore, our empirical study may not represent the general result of the usage of 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

5 Related Work

This paper aims 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.

5.1 On Using Higher-Order Functions

Many studies used higher-order functions as a new paradigm to solve complex problems. Wester and Kuper (2013) 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 (Wester and Kuper 2014). Clark and Barn (2013) used higher-order functions in dynamic reconstruction of event-driven architectures to increase the flexibility of the model. Bassoy and Schatz (2018) 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. (2016) treated services as functions and used higher-order functions to combine these services without creating new services. Racordon (2018) leveraged higher-order functions to implement components to provide coroutines for programming language without coroutines.

Existing studies have been conducted experiments to understand the difficulty of verifying and testing higher-order functions. Madhavan et al. (2017) 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. (2015) presented a validator for pure higher-order functional Scala programs, which support arbitrary function types and nested anonymous functions. Rusu and Arusoaie (2017) embedded a higher-order functional language with imperative features into the Maude framework to verify higher-order functional programs. Selakovic et al. (2018) presented LambdaTester to automate test generation for higher-order functions in dynamic languages. Xu et al. (2019) designed an automated method to identify potential calls of higher-order functions for Scala programs. Lincke and Schupp (2012) 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.

5.2 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 (2017) presented a Scala library called ScalaSMT, which supports the Satisfiability Modulo Theory (SMT) solving in Scala via accessing mainstream SMT solvers. Kroll et al. (2017) 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 (2017) designed a Scala framework that can be used for experimenting with super-compilation techniques and constructed directly from an interpreter. Reynders et al. (2018) 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 (2018) presented the first implemented design for records in Scala which enables type-safe record operations. Rahman et al. (2020) conducted a study on code clone detection on Scala programs. In the field of education, van der Lippe et al. (2016) leveraged the Scala programming language and the WebLab online learning management system to automate specification tests on the submissions by students. 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.

6 Conclusions

Constructing higher-order functions from existing functions decreases the number of similar functions and improves the code reusability. Understanding higher-order functions can help support automated code reusability and code generation. 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 35 Scala projects with the most stars. Our study shows that 6.84% of functions in these 35 Scala programs are defined as higher-order functions and 47.31% of higher-order functions are defined and called by the same developers. Meanwhile, we found that the number of calls of higher-order functions highly correlates with the code complexity of function definitions. Results in this study can be used to support the use of higher-order functions in Scala programs in practice.

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 difficulties or practical issues in using higher-order functions. Meanwhile, we plan to investigate when to use higher-order functions and analyze the cases where higher-order functions could help but were not applied. We also plan to design experiments on the changes of higher-order functions, e.g., different developers who have changed the definition of higher-order functions. This may help understand the evolution of higher-order functions and guide future development with higher-order functions. Another future work is to investigate higher-order functions in other programming languages. We aim to understand the differences in usage patterns of higher-order functions between Scala and other languages.

Acknowledgements The work is supported by the National Key R&D Program of China under Grant No. 2018YFB1003901, the National Natural Science Foundation of China under Grant No. 61872273, and the Advance Research Projects of Civil Aerospace Technology – Communications, Navigation and Remote Sensing Integrated Applications and Multi-source Spatial Data Fusion Technology.

References

Albrecht AJ Jr, Gaffney JE (1983) Software function, source lines of code, and development effort prediction: A software science validation. IEEE Trans Softw Eng 9(6):639–648. https://doi.org/10.1109/TSE.1983.235271

Altenkirch T (2001) Representations of first order function types as terminal coalgebras. In: Proceedings of the 5th International Conference on Typed Lambda Calculi and Applications, TLCA 2001, Krakow, pp. 8–21. https://doi.org/10.1007/3-540-45413-6_5

Bacchelli A, Bird C (2013a) Expectations, outcomes, and challenges of modern code review. In: Proceedings of the 2013 International Conference on Software Engineering. IEEE Press, pp 712–721

Bacchelli A, Bird C (2013b) Expectations, outcomes, and challenges of modern code review. In: Notkin D, Cheng BHC, Pohl K (eds) 35th International Conference on Software Engineering, ICSE '13. IEEE Computer Society, San Francisco, pp 712–721. https://doi.org/10.1109/ICSE.2013.6606617

Bassoy C, Schatz V (2018) Fast higher-order functions for tensor calculus with tensors and subtensors. In: Proceedings of the 18th International Conference on Computational Science, ICCS 2018, Wuxi, Proceedings, Part I, pp 639–652. https://doi.org/10.1007/978-3-319-93698-7_49

Brachthäuser JI, Schuster P (2017) Effekt: extensible algebraic effects in scala. In: Proceedings of the 8th ACM SIGPLAN International Symposium on Scala, SCALA@SPLASH 2017, Vancouver, pp 67–72. https://doi.org/10.1145/3136000.3136007

Budtz-Jorgensen E, Keiding N, Grandjean P, Weihe P (2007) Confounder selection in environmental epidemiology: Assessment of health effects of prenatal mercury exposure. Ann Epidemiol 17:27–35. https://doi.org/10.1016/j.annepidem.2006.05.007

Cardelli L, Wegner P (1985) On understanding types, data abstraction, and polymorphism. ACM Comput Surv 17(4):471–522. https://doi.org/10.1145/6041.6042

Cassez F, Sloane AM (2017) Scalasmt: satisfiability modulo theory in scala. In: Proceedings of the 8th ACM SIGPLAN International Symposium on Scala, SCALA@SPLASH 2017, Vancouver, pp 51–55. https://doi.org/10.1145/3136000.3136004

Clark T, Barn BS (2013) Dynamic reconfiguration of event driven architecture using reflection and higher-order functions. Int J Softw Inf 7(2):137–168. http://www.ijsi.org/ch/reader/view_abstract.aspx?file_no=i157

Cohen J, Cohen P, West SG, Aiken LS (2013) Applied multiple regression/correlation analysis for the behavioral sciences. Routledge, Abingdon

Fry T, Dey T, Karnauch A, Mockus A (2020) A dataset and an approach for identity resolution of 38 million author ids extracted from 2b git commits. arXiv:2003.08349

Gousios G, Storey MD, Bacchelli A (2016) 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, pp 285–296. https://doi.org/10.1145/2884781.2884826

Gu Y, Xuan J, Zhang H, Zhang L, Fan Q, Xie X, Qian T (2019) Does the fault reside in a stack trace? assisting crash localization by predicting crashing fault residence. J Syst Softw 148:88–104. https://doi.org/10.1016/j.jss.2018.11.004

HackerNews (2009) Twitter jilts Ruby for Scala. http://news.ycombinator.com/item?id=542716

Karlsson O, Haller P (2018) Extending scala with records: design, implementation, and evaluation. In: Proceedings of the 9th ACM SIGPLAN International Symposium on Scala, SCALA@ICFP 2018, St. Louis, pp 72–82. https://doi.org/10.1145/3241653.3241661

Koch R (2011) The 80/20 Principle: The Secret of Achieving More with Less: Updated 20th anniversary edition of the productivity and business classic. Hachette, UK

Kroll L, Carbone P, Haridi S (2017) 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, pp 73–77. https://doi.org/10.1145/3136000.3136009

Lee PH (2015) Should we adjust for a confounder if empirical and theoretical criteria yield contradictory results? A simulation study Scientific Reports 4(6085). https://doi.org/10.1038/srep06085

Lincke D, Schupp S (2012) 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, pp 3. https://doi.org/10.1145/2427048.2427051

Madhavan R, Kulal S, Kuncak V (2017) 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, pp 330–343. http://dl.acm.org/citation.cfm?id=3009874

McCabe TJ (1976) A complexity measure. IEEE Trans Softw Eng 2(4):308–320. https://doi.org/10.1109/TSE.1976.233837

McIntosh S, Kamei Y, Adams B, Hassan AE (2016) An empirical study of the impact of modern code review practices on software quality. Empir Softw Eng 21(5):2146–2189. https://doi.org/10.1007/s10664-015-9381-9

Nakaguchi T, Murakami Y, Lin D, Ishida T (2016) Higher-order functions for modeling hierarchical service bindings. In: IEEE International conference on services computing, SCC 2016, San francisco, pp 798–803. https://doi.org/10.1109/SCC.2016.110

Nystrom N (2017) A scala framework for supercompilation. In: Proceedings of the 8th ACM SIGPLAN International Symposium on Scala, SCALA@SPLASH 2017, Vancouver, pp 18–28. https://doi.org/10.1145/3136000.3136011

Odersky M, Altherr P, Cremet V, Emir B, Maneth S, Micheloud S, Mihaylov N, Schinz M, Stenman E, Zenger M (2004) An overview of the scala programming language. Tech. rep., Technical Report IC/2004/64. EPFL Lausanne, Switzerland

Racordon D (2018) Coroutines with higher order functions. CoRR arXiv:1812.08278

Rahman W, Xu Y, Pu F, Xuan J, Jia X, Basios M, Kanthan L, Li L, Wu F, Xu B (2020) Clone detection on large scala codebases. In: IEEE 14Th international workshop on software clones, IWSC 2020. IEEE, London, pp 38–44. https://doi.org/10.1109/IWSC50091.2020.9047640

Reed WJ (2001) The pareto, zipf and other power laws. Econ Lett 74(1):15–19

Reynders B, Greefs M, Devriese D, Piessens F (2018) 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, pp 69–74. https://doi.org/10.1145/3191697.3191731

Richardson B (2017) When should i use higher order functions? http://www.quora.com/When-should-I-use-higher-order-functions

Richmond D, Althoff A, Kastner R (2018) Synthesizable higher-order functions for C++. IEEE Trans CAD Integr Circ Syst 37(11):2835–2844. https://doi.org/10.1109/TCAD.2018.2857259

Rigby PC, Storey MD (2011) Understanding broadcast based peer review on open source software projects. In: Taylor RN, Gall HC, Medvidovic N (eds) Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011. ACM, Waikiki, pp 541–550. https://doi.org/10.1145/1985793.1985867

Rusu V, Arusoaie A (2017) Executing and verifying higher-order functional-imperative programs in maude. J Log Algebr Meth Program 93:68–91. https://doi.org/10.1016/j.jlamp.2017.09.002

Scala (2020) The scala language. http://scala-lang.org/

Selakovic M, Pradel M, Karim R, Tip F (2018) Test generation for higher-order functions in dynamic languages. PACMPL 2(OOPSLA):161:1–161:27. https://doi.org/10.1145/3276531

van der Lippe T, Smith T, Pelsmaeker D, Visser E (2016) 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, pp 65–74. https://doi.org/10.1145/2998392.2998402

Voirol N, Kneuss E, Kuncak V (2015) Counter-example complete verification for higher-order functions. In: Proceedings of the 6th ACM SIGPLAN Symposium on Scala, Scala@PLDI 2015, Portland, pp 18–29. https://doi.org/10.1145/2774975.2774978

Walpole RE, Myers SL, Ye K, Myers RH (2007) Probability and statistics for engineers and scientists. Pearson, London

Wester R, Kuper J (2013) A space/time tradeoff methodology using higher-order functions. In: 23Rd international conference on field programmable logic and applications, FPL 2013, Porto, pp 1–2. https://doi.org/10.1109/FPL.2013.6645613

Wester R, Kuper J (2014) Design space exploration of a particle filter using higher-order functions. In: Reconfigurable Computing: Architectures, Tools, and Applications - 10th International Symposium, ARC 2014, Vilamoura. Proceedings, pp 219–226. https://doi.org/10.1007/978-3-319-05960-0_21

Wilcoxon F (1992) Individual comparisons by ranking methods. In: Breakthroughs in statistics. Springer, pp 196–202

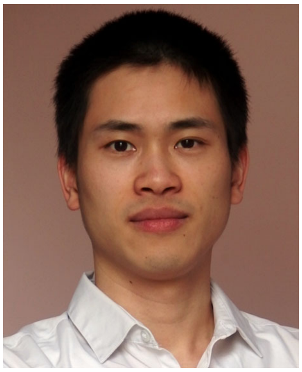
Xu Y, Jia X, Xuan J (2019) Writing tests for this higher-order function first: Automatically identifying future callings to assist testers. In: Proceedings of the 11th Asia-Pacific Symposium on Internetware (Internetware 2019), Fukuoka, pp 1–10. https://doi.org/10.1145/1122445.1122456

Zhang X, Chen Y, Gu Y, Zou W, Xie X, Jia X, Xuan J (2018) 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, pp 228–239. https://doi.org/10.1109/ICSME.2018.00032

Zhao Y, Serebrenik A, Zhou Y, Filkov V, Vasilescu B (2017) The impact of continuous integration on other software development practices: a large-scale empirical study. In: Rosu G, Penta MD, Nguyen TN (eds) Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017. IEEE Computer Society, Urbana, pp 60–71. https://doi.org/10.1109/ASE.2017.8115619

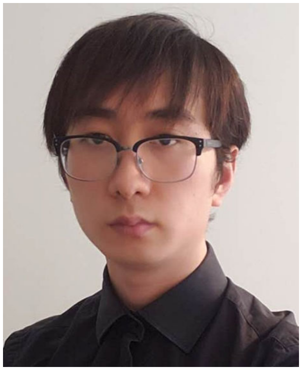
Zou W, Xuan J, Xie X, Chen Z, Xu B (2019) How does code style inconsistency affect pull request integration? an exploratory study on 117 github projects. Empir Softw Eng 24(6):3871–3903. https://doi.org/10.1007/s10664-019-09720-x

Yisen Xu is a master student at the School of Computer Science, Wuhan University, China, under supervision of Prof. Jifeng Xuan. He received the bachelor degree in software engineering at School of Computer Science, Wuhan University, in 2019. His research interests lie within software testing and mining software repositories.



Fan Wu is a co-founder of Turing Intelligence Technology. He holds Ph.D. degree in Software Engineering from UCL. He is renowned for his work on Deep Optimisation on software systems, a research field he co-founded, which has drawn significant attention and grown rapidly. He serves as reviewer and Program Committee member for prestigious research conferences and journals, such as Journal of Systems and Software (JSS) on 2016, 2017 and 2018, Information Software and Technology (IST) on 2017, and The Genetic and Evolutionary Computation Conference (GECCO) on 2018. Previously he was a postgraduate researcher for Tsinghua University on CUDA optimisation, after obtaining his first degree from the same university. His research interests include search based software engineering, software genetic improvement, evolutionary computation, machine learning.

Xiangyang Jia received the Ph.D. degree in computer software and theory from Wuhan University, China, in 2008. From 2014 to 2015, he was a visiting researcher with the Dependable Evolvable Pervasive Software Engineering Group, Politecnico di Milano. He is currently a lecturer with the School of Computer Science, Wuhan University. His current research interests include symbolic execution, software analysis, search-based software engineering, and mining software repositories. Dr. Jia is a member of the CCF. He received the Hubei Science and Technology Progress Award in 2014.



Lingbo Li is a co-founder of Turing Intelligence Technology. He received Ph.D. in Software Engineering from University College London under the supervision of Prof. Mark Harman (Facebook). He was subsequently invited to take associate professorship at the School of Computer Science, Wuhan University, China. Academically, he serves on the program committee and as reviewer for various prestigious research conferences and journals, such as, Journal of Systems and Software (JSS) on 2017 and 2018, Information Software and Technology (IST) on 2017, IEEE Intelligent Systems on 2017, and the Genetic and Evolutionary Computation Conference (GECCO) on 2018, etc. His research interests include search based software engineering, requirement engineering, evolutionary computation, deep learning.

Jifeng Xuan is a professor at the School of Computer Science, Wuhan University, China. He received the BSc degree and the PhD degree from Dalian University of Technology, China. He was previously a postdoctoral researcher at the INRIA Lille-Nord Europe, France. He is a reviewer of journals and conferences, including TSE, TOSEM, TKDE, TEVC, EMSE, ICSE, and FSE. He is a member of the ACM, IEEE, and CCF. His research interests include software testing and debugging, software data analysis, and search based software engineering.

Affiliations

Yisen Xu[1] · Fan Wu[2] · Xiangyang Jia[1] · Lingbo Li[2] · Jifeng Xuan[1]

Yisen Xu
xuyisen@whu.edu.cn

Fan Wu
fan@turintech.ai

Xiangyang Jia
jxy@whu.edu.cn

Lingbo Li
lingbo@turintech.ai

1 School of Computer Science, Wuhan University, Wuhan 430072, China

2 Turing Intelligence Technology Limited, 1 Ropemaker St, London EC2Y 9ST, UK