Writing Tests for This Higher-Order Function First

Automatically Identifying Future Callings to Assist Testers

Yisen Xu
School of Computer Science
Wuhan University
Wuhan, China
xuyisen@whu.edu.cn

Xiangyang Jia
School of Computer Science
Wuhan University
Wuhan, China
jxy@whu.edu.cn

Jifeng Xuan*
School of Computer Science
Wuhan University
Wuhan, China
jxuan@whu.edu.cn

ABSTRACT

In functional programming languages, such as Scala and Haskell, a higher-order function is a function that takes one or more functions as parameters or returns a function. Using higher-order functions in programs can increase the generality and reduce the redundancy of source code. To test a higher-order function, a tester needs to check the requirements and write another function as the test input. However, due to the complexity of higher-order functions, testing higher-order functions is a time-consuming and labor-intensive task. Testers have to spend an amount of manual effort in testing all higher-order functions. Such testing is infeasible if the time budget is limited, such as a period before a project release. In this paper, we propose an automatic approach, namely Phof, which predicts whether a higher-order function will be called in the future. Higher-order functions that are most likely to be called should be tested first. Our approach can assist developers to reduce the number of higher-order functions under test. In Phof, we extracted 24 features from source code and logs to train a predictive model based on known higher-order functions calls. We empirically evaluated our approach on 2854 higher-order functions from six real-world Scala projects. Experimental results show that Phof based on the random forest algorithm and the SMOTE strategy performs well in the prediction of calls of higher-order functions. Our work can be used to support the scheduling of limited test resources.

CCS CONCEPTS

• **Software and its engineering → Functional languages**; **Maintaining software**; *Software functional properties.*

KEYWORDS

higher-order functions, function calls, test scheduling, Scala programs

ACM Reference Format:
Yisen Xu, Xiangyang Jia, and Jifeng Xuan. 2019. Writing Tests for This Higher-Order Function First: Automatically Identifying Future Callings to Assist Testers. In *Internetware '19: Proceedings of the 11th Asia-Pacific Symposium on Internetware (Internetware '19), October 28–29, 2019, Fukuoka, Japan.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3361242.3361256

*Corresponding author

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Internetware '19, October 28–29, 2019, Fukuoka, Japan

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-7701-0/19/10…$15.00
https://doi.org/10.1145/3361242.3361256

1 INTRODUCTION

A higher-order function is a function that takes a function as an input parameter or returns a function as a result. In functional programming languages, such as Scala and Haskell, using higher-order functions can increase the generality and reduce the redundancy of source code [1, 19]; meanwhile, translating specific requirements into higher-order functions can enhance the usability and ease the collaborative development [14, 20, 25].

Similar to general functions, it is inevitable to test a higher-order function to improve the quality. However, testing a higher-order function is difficult. The aim of testing a higher-order function is to execute paths and trigger hidden faults [24]. Given a higher-order function under test, a tester has to complete two steps, including checking the requirements and writing a function as the input. Due to the complexity of higher-order functions, testing higher-order functions is a time-consuming and labor-intensive task.

Testing higher-order functions has not attracted adequate attention from the test community. For manual testing of higher-order functions, testers are required to confirm the functions that serve as the input or the output [13]. For automatic testing, many automated testing tools, such as Quickcheck in Haskell,[1] only support first-order functions. This makes testing all higher-order functions infeasible.

Instead of testing all higher-order functions, we consider testing higher-order functions that will be called in the future as a workaround. We propose a predictive approach, namely Phof, which identifies whether a higher-order function will be called in the future. A tester can write test cases for these identified higher-order functions first. This can reduce the cost of testing higher-order functions in a limited time budget. In Phof, we extracted 24 features from source code and logs to train a predictive model based on known higher-order functions calls.

We conducted empirical evaluation on 2854 higher-order functions from six real-world Scala projects and answered three research questions. Experimental results show that Phof based on the random forest algorithm and the SMOTE strategy perform well in the prediction of the calls of higher-order functions. We also evaluated the top-10 features in each project as well as their contributions in the prediction. Our approach can assist developers to reduce the number of higher-order functions under test.

[1] QuickCheck, http://hackage.haskell.org/package/QuickCheck.

Application scenario. Given a limited time budget, e.g., the time before a new release of the project, our approach PHOF can be used to identify whether a higher-order function can be called in the future. Then a tester can prioritize higher-order functions under test to avoid testing uncalled ones. Our work can be used to support the scheduling of limited test resources and reduce the cost by testers.

This paper makes the following major contributions:

- We proposed an automatic approach, namely PHOF, which predicts whether a higher-order function will be called in the future;
- We empirically evaluated 2854 higher-order functions from six real-world Scala projects.
- We investigated three research questions, including the effectiveness, the imbalance data processing strategies, and the dominant features.

The rest of this paper is organized as follows. Section 2 shows the background and motivation of studying the prediction of higher-order functions calls. Section 3 presents the proposed approach in our work. Section 4 presents the study setup, including three research questions and the data preparation. Section 5 describes the results of our exploratory study. Section 6 discusses the threats to the validity. Section 7 lists the related work and Section 8 concludes.

2 BACKGROUND AND MOTIVATION

We present the background and the motivation of our work.

2.1 Background

Scala, a programming language that supports both object-oriented programming and functional programming, is designed to make up for the deficiencies in the Java language. Scala has a strong static type system and shares many features of functional programming languages with Standard ML and Haskell, including currying, type inference, and immutability [21, 28]. Source code written in Scala is compiled into bytecode and run on a virtual machine. For the compatibility, Scala programs can directly interact with Java programs.

As a feature of functional programming, higher-order functions are directly supported in Scala. In a higher-order function, a function can be used as an input parameter and used as an output. Higher-order functions make the Scala language be used in many scenarios, including constructing distributed systems and web projects. For instance, Nystrom et al. [20] has presented a Scala framework for experimenting with super-compilation techniques; Twitter has constructed several infrastructures in Scala [8].

Figure 1 shows an excerpt of a real-world higher-order function `subName()` in Project `scala/scala`.[2] The higher-order function `subName()`, defined inside another function `scala.tools.nsc.symtab.classfile.ClassfileParser.sigToType()`, is designed to get a subset of Class `Name`. Class `Name` has the same functions and fields as Class `String`, such as the method `charAt()` and the field `length`. The definition of this higher-order function contains one input parameter and a return type. The only input parameter `isDelimiter()` at Line 5 is a first-order function, which receives a

```
1  private def sigToType(sym: Symbol, sig: Name): Type = {
2      var index = 0
3      val end = sig.length
4      ...
5      def subName(isDelimiter: Char => Boolean): Name = {
6          val start = index
7          while (! isDelimiter(sig.charAt(index))) { index += 1 }
8          sig.subName(start, index)
9      }
10     ...
11 }
```

Figure 1: Excerpt of a real-world higher-order function `subName()` from Class `scala.tools.nsc.symtab.classfile.ClassfileParser` in Project `scala/scala`.

`Char` object as input and returns a `Boolean` object. The parameter `isDelimiter()` is called at Line 7 to determine whether the `Char` object is a delimiter. The return type of the function `subName()` at Line 5 is a `Name` object. The return statement of the function `subName()` locates at Line 8.

Testing is an important phase to improve the quality of source code. ScalaTest, like JUnit in Java, is a testing framework for Scala and Java testers.[3] Many popular Scala projects, including the six projects in our study, have deployed ScalaTest to support test management and execution.

2.2 Motivation

Testing a higher-order function is difficult. To manually write a test case for a higher-order function, a tester needs to check the requirements of the function and then write another function as the input for the higher-order function under test. However, for the input of a test case, creating a function parameter is different from creating a primitive parameter (e.g., an integer or a floating-point number) or an object parameter (an object of a class). The definition of a function is various. In a study by Selakovic et al. [24], test cases that are generated by four test generation methods can reach a low ratio of code coverage in testing higher-order functions in Javascript. In their study, several basic higher-order functions are barely tested, including higher-order functions of `filter()`, `map()`, and `then()`.

Resources for testing are limited [5]. Due to the complexity of higher-order functions, testing higher-order functions is a time-consuming and labor-intensive task. In a limited time budget, such as the period before a release, it is infeasible to test all higher-order functions in a project. Instead of testing all higher-order functions, we consider testing higher-order functions that will be called in the future.

Motivated by the cost of testing higher-order functions, we propose a new approach, namely PHOF, which predicts whether a higher-order function will be called in the future. These higher-order functions that are most likely to be called can be tested first to save the cost. Our approach can assist testers to reduce the number

[2] Project scala/scala, http://github.com/scala/scala/.

[3] ScalaTest, http://scalatest.org/.

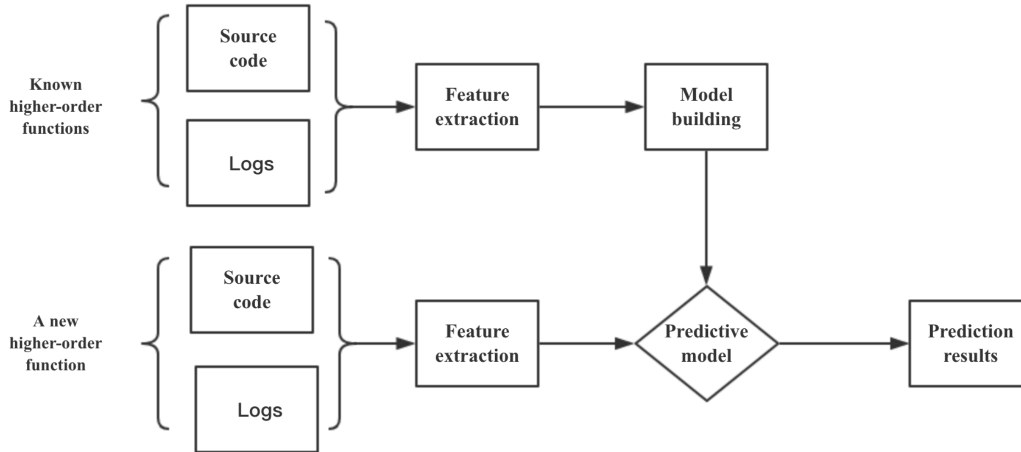

Figure 2: Overview of PHOF, an automated approach to predicting whether a higher-order function will be called in the future.

of higher-order functions under test and reduce the cost of testing higher-order functions.

3 PREDICTING CALLINGS FOR HIGHER-ORDER FUNCTIONS

We presents the overview, the feature extraction, and the learning algorithm in our proposed approach.

3.1 Overview

We designed PHOF, short for Prediction for Higher-Order Functions, an automatic approach for predicting whether a higher-order function will be called in the future. The problem of such prediction can be viewed as a classification problem with binary labels: called or uncalled. A higher-order function is labeled as **called** if there exist one or more callings while a higher-order function is labeled as **uncalled** if there are no callings for the function. A tester can then use our approach to prioritize higher-order functions to save the cost of testing within a limited time budget.

Figure 2 shows the overview of our proposed approach, PHOF. PHOF is designed to predict whether a higher-order function will be called in the future. The input of PHOF is source code and logs of a Scala project. The output of PHOF is the binary predicted result of a new higher-order function. To build a predictive model in PHOF, we extracted 24 features from source code and logs. Then, a predictive model is built based on higher-order functions with known callings and is used to predict results for new higher-order functions.

3.2 Feature Extraction

To build our model, we extracted 24 features from source code and logs of Scala programs. These 24 features are divided into three groups: Group CS – 16 features related to code statements (CS01 to CS16), Group CP – 5 features related to function properties (CP01 to CP05), and Group CG – 3 features extracted from git logs (CG01 to CG03). Table 1 list the 24 features in PHOF. In Group CS, we used the 16 features to represent the structure information of higher-order functions. In Group CP, features related to function properties, such as the Cyclomatic complexity and the executable lines of code, are used to reveal the overall state of a higher-order function. In Group CG, we distilled three features from commits logs since the information of commits may be critical for the calls of higher-order functions.

Given the source code of a Scala project, we extracted features of Group CS and Group CP by converting source files into Abstract Synthetic Trees (ASTs). Then we traversed ASTs to collect features related to code statements and function properties, such as the number of `for` statement, the Executable Lines of Code (eLoC), and the Cyclomatic complexity of higher-order functions. The *Cyclomatic complexity* is a software metric of linearly independent paths [18]. We also counted the number of code style warnings in a higher-order function (see Section 4.1 for implementation).

For Group CG, i.e., features related to logs, we extracted logs from the version control system and collected all historical commits that related to changes of higher-order functions. Then we traversed these commits and collected features of authors and commits [27].

3.3 Learning Algorithms

PHOF uses a classification algorithm to build a predictive model. We evaluated four algorithms: RandomForest – an ensemble algorithm based on decision trees, CART – a classification and regression tree, SVM – a support vector machine algorithm, and MLP – a multi-layer perceptron algorithm.

In machine learning, a decision tree is a typical classifier. Each branch represents the outcome of a classification determination and each leaf represents the classification result. RandomForest is a classifier that contains multiple decision trees, and the category of its output is determined by the mode of the output category of individual trees [2]. SVM is a binary classifier, whose purpose is to find a hyperplane to segment samples [6]. The principle of segmentation is to maximize the intervals and finally transform it into a convex quadratic programming problem. MLP is a forward-structured artificial neural network that maps a set of input vectors to a set of output vectors.

Table 1: Summary of 24 extracted features in three groups in PHOF.

Feature	Description
*Group CS – **features related to code statements***	
CS01	Whether the higher-order function has primitive types of parameters
CS02	Whether the higher-order function has a parameter that is a higher-order function
CS03	Whether the higher-order function has generic parameters, such as the parameter `A` in a class definition `class Stack[A]`
CS04	Whether the higher-order function has a return statement
CS05	Number of parameters in the definition of the higher-order function
CS06	Number of `for` statements in the higher-order function
CS07	Number of `while` statements in the higher-order function
CS08	Number of `else` statements in the higher-order function
CS09	Number of `match` statements in the higher-order function
CS10	Number of `if` statements in the higher-order function
CS11	Number of `assign` statements in the higher-order function
CS12	Number of lambda expressions in the higher-order function
CS13	Number of `try` statements in the higher-order function
CS14	Number of `apply` statements in the higher-order function
CS15	Number of local variables that can be changed in the higher-order function
CS16	Number of local variables that cannot be changed in the higher-order function
*Group CP – **features related to function properties***	
CP01	Executable lines of code (eLoC) of the higher-order function
CP02	Cyclomatic complexity of the higher-order function
CP03	Number of style warnings in the higher-order function
CP04	Containing input functions or output functions in the higher-order function (three values: functions only as input, functions only as output, and functions as both input and output)
CP05	Modifier of the higher-order function (`public`, `protected`, `private`, or `default`)
*Group CG – **features related to logs***	
CG01	Name of the first author of the higher-order function
CG02	Number of commits to source code of the higher-order function
CG03	Number of authors of the higher-order function

The distribution of **called** and **uncalled** higher-order functions is imbalanced. For most of the machine learning algorithms, the issue of data imbalance may cause incorrect prediction results [11]. Thus, we adopted the Synthetic Minority Oversampling TEchnique processing (SMOTE) strategy to address the data imbalance issue. The SMOTE strategy is a typical oversampling technique [4] of analyzing the minority samples and adding new samples to the data set according to the minority samples to achieve the data balance.

4 EXPERIMENTAL SETUP

In this section, we introduce the data preparation and the design of three research questions.

4.1 Data Preparation

Our study aims to build a learning model to predict whether a higher-order function will be called in the future, i.e., called or uncalled. We mined six Scala projects and extracted features for the construction of classifiers. Experimental results are publicly available.[4]

Our work is to train a learning model, which requires sufficient data of higher-order functions. Thus, we select six widely-used and open-sourced Scala repositories. All these projects are highly stared on GitHub and contain a large number of higher-order functions. We considered that a project with many stars indicates that the quality of the project is identified by many developers. Table 2 lists the summary of six Scala projects in the study.

We employed the static analysis tool SemanticDB to extract semantic structure, such as types and function signatures. *SemanticDB* is a Scala library to analyze and compile Scala source code.[5] We leveraged SemanticDB to extract definitions and callings of functions. First, we used SemanticDB to create a semantic database for each project. Second, we collected the definitions of higher-order functions from the semantic database if the parameters or return values of the definition contain a function. Third, we filtered out

[4] PHOF, http://cstar.whu.edu.cn/p/phof/.

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

Table 2: Summary of six Scala projects in the study. For the sake of space, each project will be denoted by its abbreviation in following sections.

Project	Abbr.	#Stars	eLoC	#Higher-order functions	Project description
scala/scala	scala	11.4k	143.6k	829	The Scala programming language
playframework/playframework	framework	11.0k	41.5k	109	A web framework for building scalable applications with Java and Scala
scalaz/scalaz	scalaz	4.1k	35.4k	1219	A Scala library for functional programming
sbt/sbt	sbt	3.8k	34.9k	264	A building tool for Scala, Java, and other languages
lampepfl/dotty	dotty	3.3k	387.1k	204	A Scala compiler
twitter/scalding	scalding	3.1k	29.6k	229	A Scala API for a Java tool named cascading
Total		36.7k	672.1k	2854	

override higher-order functions that are defined by default to implement an abstract function in a super class in the Scala language. The reason for such filtering is as follows, it is required to implement an abstract function with an override function in a sub class, but calling this override function is not required. Fourth, we extracted callings of higher-order functions by matching the function definitions. Based on these steps, we are able to extract definitions and calls of all higher-order functions in each project.

We leveraged static analysis tools Scalamata[6] and ScalaStyle[7] to extract features in Group CS and Group CP (Section 3.2). *Scalameta* is a Scala library to parse Scala files and construct ASTs. *ScalaStyle* is an off-the-shelf checking tool of the code style; we leveraged it to extract the number of code style warnings in higher-order functions. For features in Group CG, we first used the Git API to extract Git logs and collected all historical commits that related to the changes of higher-order functions [29]. Then we traversed these commits and collected the author information, including names, e-mails, and timestamps of all commits that related to higher-order functions [16]. Then we extracted the first author of the higher-order function according to the timestamp.

We labeled all higher-order functions in two categories: **called** and **uncalled**. Figure 3 presents the distribution of higher-order functions in both categories of each project. Three out of six projects have imbalanced distributions of categories, including Projects `Scala`, `sbt`, and `dotty`.

In our experiment, we used Scikit-learn for the implementation of machine learning algorithms. *Scikit-learn* is an off-the-shelf library in Python for machine learning.[8]

4.2 Evaluation Metrics

In the evaluation, we used four metrics, including precision, recall, F-measure, and accuracy. We define the four evaluation metrics based on True Positive (TP), False Positive (FP), True Negative (TN), and False Negative (FN). We list the definitions as follows,

- TP: # of higher-order functions in Category **called** that are predicted as **called**;
- FP: # of higher-order functions in Category **uncalled** that are predicted as **called**;
- TN: # of higher-order functions in Category **uncalled** that are predicted as **uncalled**;
- FN: # of higher-order functions in Category **called** that are predicted as **uncalled**.

[6]Scalameta, http://scalameta.org/.
[7]ScalaStyle, http://scalastyle.org/.
[8]Scikit-learn, http://scikit-learn.org/stable/.

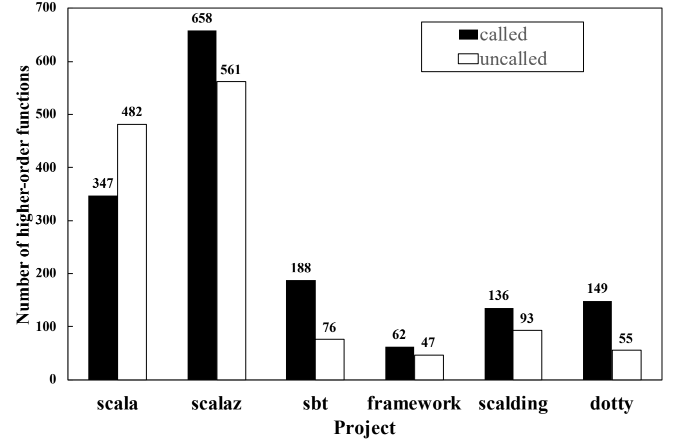

Figure 3: Distribution of called and uncalled higher-order functions in each project.

Then we define the precision, recall, F-measure, and accuracy as follows,

$$Precision(\text{called}) = \frac{TP}{TP + FP}, \; Precision(\text{uncalled}) = \frac{TN}{TN + FN}$$

$$Recall(\text{called}) = \frac{TP}{TP + FN}, \; Recall(\text{uncalled}) = \frac{TN}{TN + FP}$$

$$F\text{-}measure(\text{called}) = \frac{2 \times Precision(\text{called}) \times Recall(\text{called})}{Precision(\text{called}) + Recall(\text{called})}$$

$$F\text{-}measure(\text{uncalled}) = \frac{2 \times Precision(\text{uncalled}) \times Recall(\text{uncalled})}{Precision(\text{uncalled}) + Recall(\text{uncalled})}$$

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

4.3 Research Questions

We designed three Research Questions (RQs) to analyze the effectiveness of our proposed approach, including the prediction the callings of higher-order functions, the imbalance data processing strategies, and the feature correlation of callings of higher-order functions.

RQ1. How effective is our approach in predicting whether a higher-order function will be called in future?

We build a predictive model to predict whether a higher-order function can be called in the future. Machine learning algorithms

Table 3: Precision, recall, F-measure, and accuracy of prediction results for each of six project.

Project	Algorithm	called			uncalled			Accuracy
		Precision	Recall	F-measure	Precision	Recall	F-measure	
scala	RandomForest	0.549	0.556	0.557	**0.581**	0.330	0.410	**0.558**
	DecisionTree	0.471	**0.714**	**0.564**	0.297	0.193	0.231	0.453
	SVM	0.530	0.581	0.542	0.519	**0.463**	**0.474**	0.521
	MLP	**0.644**	0.476	0.489	0.441	0.169	0.378	0.499
scalaz	RandomForest	0.720	**0.593**	0.650	**0.705**	0.654	0.678	0.680
	DecisionTree	0.671	0.442	0.488	0.564	0.725	0.605	0.584
	SVM	**0.753**	0.579	**0.652**	0.655	**0.803**	**0.720**	**0.691**
	MLP	0.703	0.538	0.649	0.559	0.436	0.553	0.637
sbt	RandomForest	0.813	0.618	0.694	**0.718**	0.762	0.735	**0.738**
	DecisionTree	**0.876**	0.378	0.479	0.602	**0.900**	0.708	0.639
	SVM	0.757	0.571	0.649	0.662	0.825	0.734	0.698
	MLP	0.812	**0.644**	**0.745**	0.709	0.831	**0.747**	0.617
framework	RandomForest	0.763	0.579	0.651	**0.726**	0.578	0.637	**0.692**
	DecisionTree	**0.800**	0.223	0.343	0.566	**0.863**	**0.722**	0.612
	SVM	0.584	0.649	0.608	0.650	0.563	0.591	0.606
	MLP	0.704	**0.688**	**0.701**	0.679	0.740	0.684	0.670
scalding	RandomForest	0.707	0.507	**0.584**	**0.707**	0.507	0.584	**0.651**
	DecisionTree	**0.867**	0.096	0.171	0.522	**0.986**	**0.682**	0.541
	SVM	0.563	0.551	0.559	0.581	0.597	0.583	0.574
	MLP	0.662	**0.567**	0.568	0.611	0.620	0.618	0.454
dotty	RandomForest	0.769	0.578	0.658	**0.737**	0.752	0.740	0.695
	DecisionTree	0.542	0.511	0.523	0.617	0.711	0.638	0.611
	SVM	0.789	0.579	0.664	0.658	**0.826**	0.731	**0.702**
	MLP	**0.808**	**0.752**	**0.773**	0.705	0.793	**0.745**	0.627

play an important role in the prediction. We evaluate and analyze the effectiveness of four machine learning algorithms in RQ1.

RQ2. Can imbalanced data processing strategies improve prediction results?

The data imbalance of two categories of higher-order functions may lead to inaccurate classification [11]. We examine whether imbalanced data processing strategies can improve the prediction. Thus, we analyzed the impact of the SMOTE strategy and two other imbalanced data processing strategies in RQ2.

RQ3. Which features are more impactful on the prediction results?

Higher-order functions are expected to abstract the usage patterns of functions [12]. The features in Phof are divided into three groups: code statements, function properties, and logs. In RQ3, we analyzed which features affect the callings of higher-order functions.

5 EMPIRICAL RESULTS

In this section, we present and analyze the results of three RQs in our study. The result and findings are listed as follows.

5.1 RQ1. How effective is our approach in predicting whether a higher-order function will be called in future?

We aim to analyze the effectiveness of our proposed approach Phof. Four classification algorithms are used in Phof, including RandomForest, CART, SVM, and MLP. In RandomForest, the number of decision tree is set by default to 6; in CART, the parameter criterion is set to entropy; in SVM, the linear kernel is used inside the algorithm; in MLP, the number of hidden layers is set to 100. The SMOTE strategy is combined with each classification algorithm to eliminate the risk of data imbalance. We used 5-fold cross validation to evaluate the effectiveness of the experiment. For each project, we randomly divide the higher-order functions into five equal-sized folds. Then we built five rounds of experiments. In each round, one fold is used as a test set and the other four folds are used as a training set. Then the average of five rounds is reported as the result.

Table 3 presents the prediction results of each of six projects. Among four algorithms under evaluation, no algorithm can completely beat the others for all projects. In six, RandomForest achieves the highest accuracy in four projects, i.e., scala, sbt, framework, and scalding; SVM achieves the highest accuracy in two projects. For the F-measure values, all four algorithms can obtain the highest values. In Projects sbt and dotty, MLP obtained the highest values of F-measure for both **call** and **uncalled** higher-order functions; in Project scalaz, SVM obtained the highest values of F-measure. For **called** higher-order functions, DecisionTree obtains the highest precision values in three projects while MLP obtains the highest recall values in four projects. For **uncalled** higher-order functions, RandomForest obtains the highest precision values in all six projects while DecisionTree and SVM obtain the highest recall values in three projects, respectively.

As shown in Table 3, RandomForest with SMOTE achieves the highest values of accuracy in most of projects. Then we counted the

Table 4: Average of precision, recall, F-measure, and accuracy of prediction results for all the six projects.

Algorithm	called			uncalled			Accuracy
	Precision	Recall	F-measure	Precision	Recall	F-measure	
RandomForest	0.720	0.572	0.632	**0.696**	0.597	0.631	**0.670**
DecisionTree	0.705	0.394	0.428	0.527	**0.730**	0.598	0.573
SVM	0.663	0.585	0.612	0.621	0.679	**0.639**	0.632
MLP	**0.722**	**0.611**	**0.646**	0.617	0.598	0.621	0.584

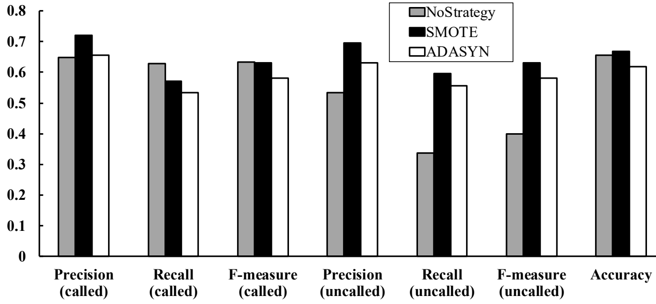

Figure 4: Average results of the random forest algorithm using three strategies of imbalanced data processing.

Table 5: Lists of top-10 dominant features for each project

Project	scala	scalaz	sbt	framework	scalding	dotty
1	**CP02** †	**CP01**	**CP01**	**CP05**	**CP05**	CS05
2	CG03	**CP02**	**CP02**	CG01	**CS10**	CS04
3	CS11	**CS09**	CS03	CS16	**CP02**	CS01
4	**CP03**	CS12	CS16	CS11	**CP03**	**CP01**
5	CS02	CS16	**CS09**	CS14	CS03	**CP05**
6	CP04	CS14	CS14	CG02	**CS09**	**CS09**
7	CG01	**CS10**	**CS10**	CG03	CS08	CS08
8	**CS10**	CS03	CP04	CP04	CS15	**CP02**
9	**CP01**	**CP05**	CS01	CS05	CS05	CS11
10	**CS09**	**CP03**	CS04	**CP03**	CG02	CS02

† We labeled features that appear for four times or more in bold.

average of six projects in Table 4. As shown in Table 4, RandomForest obtains the best accuracy and the best precision for **uncalled** higher-order functions; SVM achieves the best F-measure for **uncalled** ones. All algorithms except SVM achieve the precision over 0.700 for **called** higher-order functions. MLP achieves the highest precision of 0.722, the highest recall of 0.611, and the highest F-measure of 0.646 for **called** higher-order functions. From Table 4 and Table 3, we find that the evaluation values of four algorithms differ greatly. RandomForest performs well in the accuracy while MLP performs well in the precision and recall of **called** higher-order functions.

Finding 1. Our proposed approach, PHOF is effective in predicting whether a higher-order function will be called in the future. RandomForest as well as other algorithms, i.e., DecisionTree, SVM, and MLP, can achieve high values in the evaluation.

5.2 RQ2. Can imbalanced data processing strategies improve prediction results?

In Section 4.1, we showed that **called** and **uncalled** higher-order functions are not balanced. Thus, we evaluate the effectiveness of imbalanced data processing techniques. We used three imbalanced data processing strategies, SMOTE, ADAptive SYNthetic sampling (ADASYN), and no strategy (called *NoStrategy* for short), to solve the imbalanced problem. As mentioned in Section 3.3, the SMOTE strategy is a typical oversampling technique [4]. *ADASYN* is another typical oversampling technique [10]. The key idea of ADASYN is to weight different minority samples according to the learning difficulty of data. ADASYN can synthesize data for the minority class that are difficult to be modeled.

Figure 4 shows values of precision, reall, F-measure, and accuracy of RandomForest with three sampling strategies in six projects. As shown in the Figure 4, SMOTE achieves the maximum value in the precision for both categories, the recall and F-measure for **uncalled** higher-order functions, and the accuracy. NoStrategy achieves the maximum value in the recall and the F-measure for **called** higher-order functions while ADASYN does not achieve any maximum value in the evaluation.

Finding 2. Experiments show that the SMOTE strategy is an effective strategy of the imbalanced data processing. Compared with using no strategy, the precision and the accuracy can be improved with SMOTE.

5.3 RQ3. Which features are more impactful on the prediction results?

We tend to find out features that affect the prediction of higher-order functions callings. In each project, we use Pearson correlation coefficient to evaluate the correlation between a feature and the predicted result. Then we selected the top-10 features, which correlate the most with the prediction results. We referred to these top-10 features as *dominant features* [9].

In statistics, Pearson correlation coefficient is used to measure the degree of the linear correlation between two variables X and Y and the coefficient value is between -1 and 1 [7]. The absolute value of a coefficient is 1 if X and Y can completely linearly correlated and a coefficient is 0 if there exists no linear correlation. A positive coefficient indicates that Y increases when X increases; a negative coefficient indicates that Y decreases when X increases.

For each project in our experiment, the Pearson correlation coefficient between each feature and the predicted result are calculated; then we sorted all features according to the absolute value of Pearson correlation coefficient. Table 5 presents the lists of top-10 dominant features in each project. As shown in Table 5, CP01

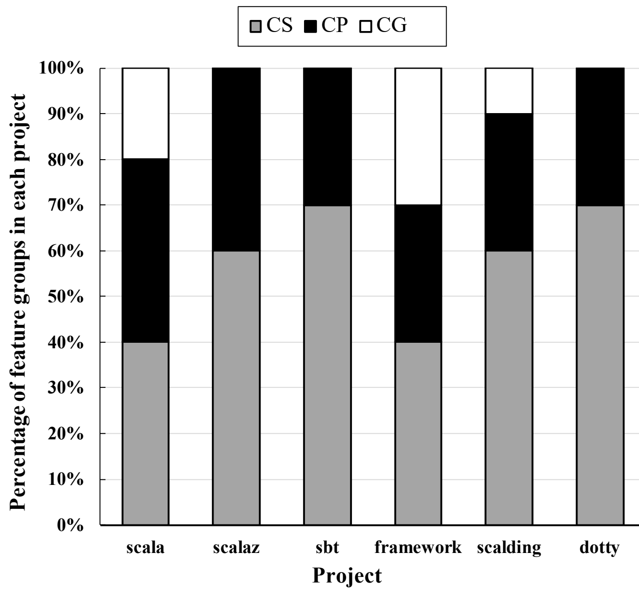

Figure 5: Percentage of feature groups in each project.

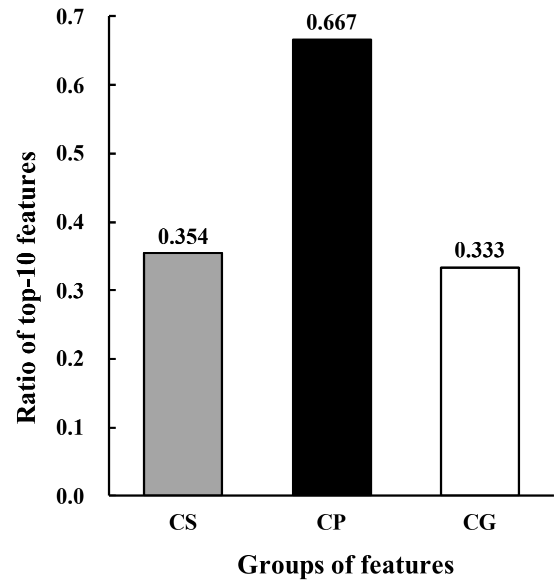

Figure 6: Ratio of dominant features in each feature group in each project.

(eLoC of higher-order functions) appears in four out of six projects, i.e., scala, scalaz, sbt, and dotty. In Projects scalaz and sbt, CP01 correlates the most with the predicted result; meanwhile, CP02 (Cyclomatic complexity of higher-order functions) and CS09 (the number of `match` statements in higher-order functions) appear in five out of six projects. CP03 (the number of style warnings in higher-order functions), CP05 (the modifier), and CS10 (the number of `if` statements) appear in four out of six projects, and CP05 shows the greatest correlation with the predicated result in Projects framework and scalding.

From Table 5, we can observe that CP01, CP02, CP03, CP05, CS09, and CS10 are highly correlative with the prediction result. As reported in Table 1, CP01, CP02, CP03, and CP05 belong to Group CP, i.e., features related to function properties; CS09 and CS10 belong to Group CS (features related to code statements). We found that the number of `match` statements (CS09) and the number of `if` statements (CS10) are used to calculate the Cyclomatic complexity of higher-order functions (CS02). This observation shows that properties of higher-order functions, such as the eLoC and the Cyclomatic complexity, have highly affected the prediction result, i.e., the callings of higher-order functions.

To analyze the features that affect the prediction result, we calculated the distribution of the top-10 dominant features in each project. Figure 5 presents the percentage of each group of features that belong to the top-10 dominant features in each project. Features of Group CS are the majority of the five projects in six projects, except for Project framework. One possible reason is that Group CS contains 16 features, which are the majority of all extracted features.

We then counted the ratio of dominant features in each feature group as follows,

$$ratio(\text{feature group}) = \frac{\text{\# of dominant features from the group}}{\text{\# of features in the group} \times \text{\# of projects}}$$

where dominant features of each project are directly extracted from Table 5. Figure 6 presents ratio of dominant features in each feature group. We can observe that the ratio of Group CP is the highest, i.e., 0.667, and the ratio of Group CG is the lowest, i.e., 0.333. This observation is consistent with the conclusion of Table 5: features in Group CP have more influence on the prediction result than features in the other two groups.

We evaluated whether the dominant features can represent all features in the prediction. Table 6 presents the comparison between the top-10 dominant features and all the 24 features. As shown in Table 6, only one out of six projects, i.e., scalding, can obtain better prediction with the dominant features than with all features. In Project scalding, all the evaluation metrics of prediction with the top-10 dominant features are better than the prediction with 24 features. In other five projects under evaluation, the accuracy with top-10 dominant features are lower than the accuracy with all 24 features.

Finding 3. Two features from the group of code statements and four features from the group of function properties highly correlate with the prediction result. Meanwhile, the top-10 dominant features can partially represent all 24 features in the prediction.

6 THREATS TO VALIDITY

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

Table 6: Comparison between the whole set of 24 features and the top-10 features on higher-order functions from six projects.

Project	Features	called			uncalled			Accuracy
		Precision	Recall	F-measure	Precision	Recall	F-measure	
scala	All 24	**0.549**	0.556	0.557	**0.581**	0.330	0.410	**0.558**
	Top-10	0.534	**0.625**	**0.574**	0.558	**0.365**	**0.438**	0.543
scalaz	All 24	**0.720**	0.593	**0.650**	**0.705**	0.654	**0.678**	**0.680**
	Top-10	0.656	**0.614**	0.633	0.643	**0.657**	0.649	0.645
sbt	All 24	**0.813**	0.618	**0.694**	**0.718**	0.762	0.735	**0.738**
	Top-10	0.763	**0.634**	0.687	0.710	**0.804**	**0.751**	0.722
framework	All 24	**0.763**	0.579	**0.651**	**0.726**	**0.578**	**0.637**	**0.692**
	Top-10	0.731	**0.596**	0.650	0.631	0.517	0.563	0.684
scalding	All 24	0.707	0.507	0.584	0.707	0.507	0.584	0.651
	Top-10	**0.757**	**0.715**	**0.734**	**0.755**	**0.758**	**0.755**	**0.744**
dotty	All 24	**0.769**	**0.578**	**0.658**	**0.737**	0.752	**0.740**	**0.695**
	Top-10	0.723	0.545	0.609	0.651	**0.764**	0.694	0.654

Threats to construct validity. In our study, we extracted 24 features from source code and logs. It is possible to design better features to characterize the prediction problem. Meanwhile, we chose four typical machine learning algorithms and three typical strategies of imbalanced data processing according to our experience. These may exist several algorithms or processing strategies that can achieve better results. The design of this work is to assist testers for better scheduling higher-order functions in manual testing. However, the evaluation does not count the effort by testers. Instead, we considered that higher-order functions that will be called later should be tested first. This provides the scenario of using our approach.

Threats to internal validity. In machine learning, the setting of parameters is important: the prediction may be hurt by the setting of parameters. In our study, we set parameters according to the API document of the Scikit-learn tool. These parameter values are not well-tuned in our dataset. A better solution to the parameter values is to conduct a large-scale experiment and tune parameter values accordingly.

Threats to external validity. Our study selected six Scala projects from GitHub. Such selection may hurt the generality of our study. We do not claim that our prediction result can be generalized to other Scala projects or even other functional languages, such as Haskell. This results in a threat to the generality.

7 RELATED WORK

We present the related work in two categories, the studying on higher-order functions and the studying on Scala programs.

7.1 Studies on Scala Programs

The Scala programming language has received much attention. Existing works have studied the Scala language and Scala programs. Reynders et al. [22] defined a multilevel language, Scalagna, which combines the existing Scala JVM and the JavaScript ecosystem into a single programming model. Nystrom [20] designed a Scala framework to implement efficient super-compilers for arbitrary programming languages. In the field of symbolic execution, Cassez and Sloane [3] proposed ScalaSMT, which supports the Satisfiability Modulo Theory (SMT) solving in Scala. In the field of education, van der Lippe et al. [25] used the Scala programming language and the WebLab online learning system to examine quizzes by students. Kroll et al. [14] proposed a framework that supports straightforward and simplified translation between formal specifications and executable code.

7.2 Studies on Higher-Order Functions

The higher-order function is a feature of the Scala language. Testing and validating higher-order functions is difficult due to the complexity of higher-order functions. To test a higher-order function, a tester has to understand the requirements and then writes an input function for the higher-order function under test. Pieter et al. [13] proposed a method to test higher-order functions by mimicking and controlling the structure of functions. This method can find errors that have not occurred in higher-order functions for several years. Selakovic et al. [24] proposed LambdaTester, which uses feedback techniques to automatically generate test cases for higher-order functions in JavaScript. For the validation of higher-order functions, Madhavan et al. [17] proposed a novel method to specify and verify the resource utilization of higher-order functional programs using lazy evaluation and memory. Voirol et al. [26] proposed a validator for pure higher-order functional Scala programs; this validator supports the validation of arbitrary function types and arbitrary nested anonymous functions. Rusu and Arusoaie [23] embedded a higher order functional language with imperative features into the Maude framework to verify higher-order functional programs. Lincke and Schupp [15] proposed a transformation that converts higher-order functions to lower-order functions by mapping higher-order types to lower-order types.

Higher-order functions are employed as a resolution for complicated problems. Bassoy and Schatz [1] used optimized higher-order functions to quickly calculate tensors, and their optimized higher-order functions achieved 68% of the maximum throughput of the

Intel Core i9-7900X. Nakaguchi et al. [19] treated services as functions and used higher-order functions to combine these services without creating new services.

Different from existing works, we designed an automated approach to the prediction of future callings of higher-order functions. Instead of directly testing higher-order functions, we identified higher-order functions that need to be tested to assist the manual testing by testers.

8 CONCLUSIONS

Manually testing a higher-order function is difficult. In this paper, we proposed an approach, namely PHOF, which can predict whether a higher-order function will be called in the future. This approach can help testers to identify higher-order functions that need to be tested first. Our work can save the time cost of testing higher-order functions in a limited time budget. Our study shows that, the random forest algorithm with the SMOTE strategy in PHOF is effective in the prediction.

In future work, we plan to extract new features such as semantic features, to enhance the prediction performance. We also plan to conduct a study to understand reasons for uncalled higher-order functions, including the reason from source code and the reason from developers or testers.

ACKNOWLEDGMENTS

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 Nos. 61872273 and 61502345, and the Technological Innovation Projects of Hubei Province under Grant No. 2017AAA125.

REFERENCES

[1] Cem Bassoy and Volker Schatz. 2018. Fast Higher-Order Functions for Tensor Calculus with Tensors and Subtensors. In *Computational Science - ICCS 2018 - 18th International Conference, Wuxi, China, June 11-13, 2018, Proceedings, Part I.* 639–652. https://doi.org/10.1007/978-3-319-93698-7_49

[2] Leo Breiman. 2001. Random Forests. *Machine Learning* 45, 1 (2001), 5–32. https://doi.org/10.1023/A:1010933404324

[3] Franck Cassez and Anthony M. Sloane. 2017. ScalaSMT: satisfiability modulo theory in Scala. In *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala, SCALA@SPLASH 2017, Vancouver, BC, Canada, October 22-23, 2017.* 51–55. https://doi.org/10.1145/3136000.3136004

[4] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. 2002. SMOTE: Synthetic Minority Over-sampling Technique. *J. Artif. Intell. Res.* 16 (2002), 321–357. https://doi.org/10.1613/jair.953

[5] Zongzheng Chi, Jifeng Xuan, Zhilei Ren, Xiaoyuan Xie, and He Guo. 2017. Multi-Level Random Walk for Software Test Suite Reduction. *IEEE Comp. Int. Mag.* 12, 2 (2017), 24–33. https://doi.org/10.1109/MCI.2017.2670460

[6] Corinna Cortes and Vladimir Vapnik. 1995. Support-Vector Networks. *Machine Learning* 20, 3 (1995), 273–297. https://doi.org/10.1007/BF00994018

[7] Leo Egghe and Loet Leydesdorff. 2009. The relation between Pearson's correlation coefficient r and Salton's cosine measure. *JASIST* 60, 5 (2009), 1027–1036. https://doi.org/10.1002/asi.21009

[8] Marius Eriksen. 2012. Effective Scala. http://twitter.github.io/effectivescala.

[9] Yongfeng Gu, Jifeng Xuan, Hongyu Zhang, Lanxin Zhang, Qingna Fan, Xiaoyuan Xie, and Tieyun Qian. 2019. Does the fault reside in a stack trace? Assisting crash localization by predicting crashing fault residence. *Journal of Systems and Software* 148 (2019), 88–104. https://doi.org/10.1016/j.jss.2018.11.004

[10] Haibo He, Yang Bai, Edwardo A. Garcia, and Shutao Li. 2008. ADASYN: Adaptive synthetic sampling approach for imbalanced learning. In *Proceedings of the International Joint Conference on Neural Networks, IJCNN 2008, part of the IEEE World Congress on Computational Intelligence, WCCI 2008, Hong Kong, China, June 1-6, 2008.* 1322–1328. https://doi.org/10.1109/IJCNN.2008.4633969

[11] Haibo He and Edwardo A. Garcia. 2009. Learning from Imbalanced Data. *IEEE Trans. Knowl. Data Eng.* 21, 9 (2009), 1263–1284. https://doi.org/10.1109/TKDE.2008.239

[12] Olof Karlsson and Philipp Haller. 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, MO, USA, September 28, 2018.* 72–82. https://doi.org/10.1145/3241653.3241661

[13] Pieter W. M. Koopman and Rinus Plasmeijer. 2006. Automatic Testing of Higher Order Functions. In *Programming Languages and Systems, 4th Asian Symposium, APLAS 2006, Sydney, Australia, November 8-10, 2006, Proceedings.* 148–164. https://doi.org/10.1007/11924661_9

[14] Lars Kroll, Paris Carbone, and Seif Haridi. 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, BC, Canada, October 22-23, 2017.* 73–77. https://doi.org/10.1145/3136000.3136009

[15] Daniel Lincke and Sibylle Schupp. 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, Estonia, March 31 - April 1, 2012.* 3. https://doi.org/10.1145/2427048.2427051

[16] Ping Ma, Danni Xu, Xin Zhang, and Jifeng Xuan. 2019. Changes Are Similar: Measuring Similarity of Pull Requests That Change the Same Code in GitHub. In *Software Engineering and Methodology for Emerging Domains*, Zheng Li, He Jiang, Ge Li, Minghui Zhou, and Ming Li (Eds.). Springer Singapore, Singapore, 115–128.

[17] Ravichandhran Madhavan, Sumith Kulal, and Viktor Kuncak. 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, France, January 18-20, 2017.* 330–343. http://dl.acm.org/citation.cfm?id=3009874

[18] Thomas J. McCabe. 1976. A Complexity Measure. *IEEE Trans. Software Eng.* 2, 4 (1976), 308–320. https://doi.org/10.1109/TSE.1976.233837

[19] Takao Nakaguchi, Yohei Murakami, Donghui Lin, and Toru Ishida. 2016. Higher-Order Functions for Modeling Hierarchical Service Bindings. In *IEEE International Conference on Services Computing, SCC 2016, San Francisco, CA, USA, June 27 - July 2, 2016.* 798–803. https://doi.org/10.1109/SCC.2016.110

[20] Nathaniel Nystrom. 2017. A Scala framework for supercompilation. In *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala, SCALA@SPLASH 2017, Vancouver, BC, Canada, October 22-23, 2017.* 18–28. https://doi.org/10.1145/3136000.3136011

[21] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. 2004. *An overview of the Scala programming language.* Technical Report. Technical Report IC/2004/64, EPFL Lausanne, Switzerland.

[22] Bob Reynders, Michael Greefs, Dominique Devriese, and Frank Piessens. 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, France, April 09-12, 2018.* 69–74. https://doi.org/10.1145/3191697.3191731

[23] Vlad Rusu and Andrei Arusoaie. 2017. Executing and verifying higher-order functional-imperative programs in Maude. *J. Log. Algebr. Meth. Program.* 93 (2017), 68–91. https://doi.org/10.1016/j.jlamp.2017.09.002

[24] Marija Selakovic, Michael Pradel, Rezwana Karim, and Frank Tip. 2018. Test generation for higher-order functions in dynamic languages. *PACMPL* 2, OOPSLA (2018), 161:1–161:27. https://doi.org/10.1145/3276531

[25] Tim van der Lippe, Thomas Smith, Daniël Pelsmaeker, and Eelco Visser. 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, Netherlands, October 30 - November 4, 2016.* 65–74. https://doi.org/10.1145/2998392.2998402

[26] Nicolas Voirol, Etienne Kneuss, and Viktor Kuncak. 2015. Counter-example complete verification for higher-order functions. In *Proceedings of the 6th ACM SIGPLAN Symposium on Scala, Scala@PLDI 2015, Portland, OR, USA, June 15-17, 2015.* 18–29. https://doi.org/10.1145/2774975.2774978

[27] Tao Wang, Yang Zhang, Gang Yin, Yue Yu, and Huaimin Wang. 2018. Who Will Become a Long-Term Contributor?: A Prediction Model based on the Early Phase Behaviors. In *Proceedings of the Tenth Asia-Pacific Symposium on Internetware, Internetware 2018, Beijing, China, September 16-16, 2018.* 9:1–9:10. https://doi.org/10.1145/3275219.3275223

[28] Yisen Xu, Fan Wu, Xiangyang Jia, Lingbo Li, and Jifeng Xuan. 2019. Mining the Use of Higher-Order Functions: An Exploratory Study on Scala Programs. In *Proceedings of the National Software Application Conference of China (NASAC 2019).* to appear.

[29] Xin Zhang, Yang Chen, Yongfeng Gu, Weiqin Zou, Xiaoyuan Xie, Xiangyang Jia, and Jifeng Xuan. 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, Spain, September 23-29, 2018.* 228–239. https://doi.org/10.1109/ICSME.2018.00032