

ABSTRACT

MATHEW, GEORGE. Cross-Language Code Similarity and Applications in Clone Detection and Code Search. (Under the direction of Kathryn T. Stolee.)

Code similarity is a precursor to multiple applications in software development, including duplicate code detection for refactoring, identifying candidates for program repair, and language translation. Most existing code similarity techniques rely on static attributes like the comparison of tokens and abstract syntax trees. These approaches have high recall, scale well to support large repositories, and many can handle cross-language analysis, but they have low precision. In contrast, contemporary dynamic similarity approaches have high precision but are not scalable and do not support comparing code across programming languages. In this research, we present three cross-language code similarity techniques to improve precision, recall, and scalability. Each approach aims to address a particular shortcoming of prior state-of-the-art and explores the costs and benefits of each approach in the applications of clone detection and code-to-code search. Our thesis: **Combining static and dynamic code similarity metrics impacts precision, recall, and scalability in code-to-code search and clone detection applications.**

We first present SLACC, a cross-language code clone detection that uses dynamic analysis. SLACC demonstrates the application of dynamic similarity to compare code across programming languages by using input/output behavior for cross-language code clone detection. Prior work on cross-language similarity, used languages with similar type systems but cross-language analyses are needed in broader contexts. SLACC is designed to target a static typed language, Java, and a dynamic typed language, Python. Compared to the state-of-the-art dynamic code clone detection tool HitoshiIO for Java, SLACC has higher precision (86.7% vs. 30.7%). SLACC is the first work to perform clone detection for dynamic typed languages (precision = 87.3%) and the first to perform clone detection across languages that lack a common underlying representation (precision = 94.1%). However, as SLACC is predicated on the availability of executable code, it has lower recall compared to static similarity approaches.

In our next study, we present COSAL, which augments SLACC with two static similarity measures. COSAL uses non-dominated search to rank similar code snippets based on code token similarity, structural similarity using a generic AST, and behavioral similarity using SLACC. COSAL is empirically evaluated on Java, Python and Haskell files from two benchmarks and find that non-dominated search on static and dynamic approaches is more effective compared to single search similarity and state-of-the-art within-language and cross-language code-to-code search tools. With respect to code clone detection, COSAL has better precision and recall compared to state-of-the-art static code clone detection tools and better recall compared to dynamic clone detection tools. However, for large code repositories, non-dominated ranking, AST-, and IO-based similarity renders COSAL impractical and less maintainable.

Our last study presents StaCE, a scalable cross-language code similarity technique using em-

bedded representations of context and structure in source code. StaCE addresses the scalability shortcomings of COSAL by fundamentally redesigning how code similarity is computed and stored. Using the static analysis components of COSAL, it learns and stores embedded representations of code snippets. Contextual features extracted from tokens of code and structural features extracted from the ASTs are used to generate the code embeddings. The embedded representation of code enables faster code search compared to non-dominated ranking. We empirically evaluate StaCE using 139,255 code snippets and find that StaCE has higher precision and recall compared to state-of-the-art code-to-code search and code-clone detection techniques.

In aggregate, this thesis presents novel contributions to the software engineering literature, with a particular focus on cross-paradigm and cross-language static and dynamic code similarity analyses. We have studied problems related to precision, recall, and scalability in an effort to produce practical, scalable, cross-language code search techniques.

© Copyright 2022 by George Mathew

All Rights Reserved

Cross-Language Code Similarity and Applications in Clone Detection and Code Search

by
George Mathew

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2022

APPROVED BY:

Timothy Menzies

Christopher Parnin

Xipeng Shen

Kathryn T. Stolee
Chair of Advisory Committee

BIOGRAPHY

The author was born and raised in Hyderabad, India. He obtained his bachelors in Electronics and Instrumentation Engineering from Amrita Vishwa Vidyapeetham, Coimbatore in 2012. He then worked for Payoda Technologies and VDP IT Technologies for two years until June 2014. He then enrolled in the masters program in Computer Science at North Carolina State University (NCSU). He received his MS in 2016 under the guidance of Dr. Tim Menzies and was funded by NASA Jet Propulsion Laboratory. Post his masters, he enrolled in the Ph.D. program in Computer Science at NCSU. During his Ph.D. he worked on Software Effort Estimation, optimizing Requirements Engineering models and developed measures to identify similar code across Programming Languages. The author will be joining Facebook as a Research Scientist at New York.

*Ever Tried, Ever Failed, No Matter.
Try Again, Fail Again, Fail Better.*

Samuel Beckett

ACKNOWLEDGEMENTS

The Ph.D. journey has been an amalgamation of many high, few low and numerous eventful phases with multiple pots of coffee and bowls of biryani. As I look back over this journey, I would be lying if I said this thesis is my work. It is a result of the sacrifices, tears and love of multiple people before and during my graduate program. I try my best to thank everyone who has been a part of this journey. However, these acknowledgments are not comprehensive.

First, I would like to express my heartfelt gratitude to my advisor, Dr. Katie Stolee. Dr. Stolee's support and guidance has always helped me and without whose ideas and contributions, this research would not be possible. I would always meander away in random directions with respect to research ideas, but Dr. Stolee ensured that we stayed on track with methodical research and breakdown complex problems into simpler smaller ones. I see my relationship with Dr. Stolee beyond a student-advisor. Dr. Stolee's perspective on life and advise on my personal life has been helpful during some of the toughest times of my Ph.D. I am equally grateful to Dr. Tim Menzies for introducing me to the world of software engineering research. I am lucky to collaborate with Dr. Menzies on a myriad of different projects where I was able to hone my skills and figure out what my true calling in research. Dr. Menzies has always inspired me to work smart, think outside the box and sense out promising areas of research. Our conversations have always been insightful, entertaining and I can say without a doubt that Dr. Menzies is one of the wisest and funniest person I know. I am where I am, since I was standing on the shoulders of these two giants.

I am incredibly grateful to my other committee members Dr. Chris Parnin and Dr. Xipeng Shen for their critiques and comments. Dr. Parnin helped in streamlining the first component of this research and without Dr. Parnin's efforts this research would be incomplete. I thank my collaborators Dr. Neil Ernst, Dr. John Klein, Dr. Jarius Hihn and Dr. Ye Yang.

I had the great privilege to share this journey with great peers through different phases of the Ph.D. program. Dr. Peipei Wang, Justin Middleton, Gina Bai and Kai Presler-Marshall have given critical feedback and inputs for this research. I thank Dr. Vivek Nair, Dr. Wei Fu, Dr. Rahul Krishna and Dr. Tianpei Xia for great conversations on research and life. Our travels across Europe are some of my most common dinner table stories. I thank all the students from RAISE and other Software Engineering labs at NCSU for their great research for pushing boundaries in Software Engineering and keeping me motivated. I thank Dr. George Rouskas, Kathy Luca, Carol Allen and all faculty and staff in the Department of Computer Science for providing a great research environment.

I have been blessed with great friends, teachers and family. This degree is a proof of their love and support. I thank my mother Mariyam Jacob for her love and sacrifices. I thank my brother Jacob Mathew for his guidance and for always being a great inspiration. I would like to remember my late father Varghese Mathew and I am grateful to him for ensuring I had a great life. I thank my partner Minita Varghese for her love, kindness and patience through this process. I thank my best friend Sandeep Mohan for his support, critiques and being constantly present. I thank my friends Sharanya Ramesh, Sumith Sashidran and Anant Parekh for keeping me motivated and always having my back.

I thank my friend Akhilesh Tanneru for being the best roommate and sharing our celebrations and solaces. I thank my friends Supriya Mohan and Sujith Perla for giving me a place to stay and inspiring me to do better. I will always be thankful to Srinivas Ramachandran and family for welcoming me into your house and making me feel at home miles away from home. I thank my cousin Ginu Easow and her family for their support and kindness. I am deeply thankful to Shantha Ramachandran and Mohan Narayanan for their optimism and supporting me since my under-graduation. A huge shout-out to my friends Nishranth, Praveen, Sairam, Mourya, Ritu, Malavika, Gopi and Nandakumar. I am extremely grateful to my teachers late Ms. Urmila Kulkarni, Mr. Rajesh KK and Dr. Binoy Nair for inspiring me and keeping me grounded.

A large part of this journey has been through the COVID-19 pandemic. I express my deepest gratitude and remember the health care and essential service workers who ensured that we could have a better life at the cost of theirs. Thank you. YNWA.

TABLE OF CONTENTS

LIST OF TABLES	viii
LIST OF FIGURES	x
Chapter 1 Introduction	1
1.1 Overview	2
1.2 Contributions	3
1.3 Outline	3
Chapter 2 Background	5
2.1 Similarity of Code	5
2.1.1 Static Code Similarity	6
2.1.2 Dynamic Code Similarity	8
2.2 Pareto Dominance	9
2.3 Embedding Objects	9
Chapter 3 Simion Based Language Agnostic Code Clones	11
3.1 Motivation	11
3.2 SLACC	13
3.2.1 Segmentation	14
3.2.2 Function Creation	14
3.2.3 Input Generation	16
3.2.4 Execution	18
3.2.5 Clone Detection	18
3.3 Research Questions	20
3.4 Data	20
3.5 Experimental Setup	21
3.6 Metrics	21
3.7 Baselines	22
3.7.1 RQ1: HitoshiIO	22
3.7.2 RQ2: Automated AST Comparison	23
3.7.3 RQ3: Manual Cross-language AST Comparison	23
3.8 Precision Analysis	24
3.9 Results	24
3.9.1 RQ1: Static Typed Languages	24
3.9.2 RQ2: Dynamic Typed Languages	27
3.9.3 RQ3: Across Programming Languages	28
3.10 Limitations and Threats	29
Chapter 4 Code-to-Code Search Across Languages	31
4.1 Motivation	31
4.2 COSAL	33
4.2.1 Token-based Search	34
4.2.2 AST-based Search	36
4.2.3 Input-Output based Search	39
4.2.4 Non-Dominated Sorting	40

4.3	Supporting functional programming	41
4.4	Research Questions	42
4.5	Data	43
4.5.1	AtCoder	43
4.5.2	BigCloneBench	44
4.6	Baselines	44
4.6.1	RQ5, RQ8: Text Search	44
4.6.2	RQ5, RQ8: GitHub Search	44
4.6.3	RQ6: FaCoY	45
4.6.4	RQ7: ASTLearner	45
4.6.5	RQ7: CLCDSA	45
4.7	Metrics	45
4.7.1	Code Search Metrics	45
4.7.2	Clone Detection Metrics	46
4.8	Experimental Setup	47
4.9	Results	47
4.9.1	RQ4: Single vs Multi-Objective Search	47
4.9.2	RQ5: State-of-the-Practice Code Search	48
4.9.3	RQ6: Single Language Code Search	49
4.9.4	RQ7: Code clone detection	50
4.9.5	RQ8: Code-to-code search for functional languages	51
Chapter 5 Structural and Contextual Embeddings		54
5.1	Motivation	54
5.2	Structural and Contextual Embeddings	56
5.2.1	Network Architecture	56
5.2.2	Pre-processing	57
5.2.3	Vectorizing Structure	57
5.2.4	Vectorizing Context	58
5.2.5	Vectorizing Code	59
5.2.6	Training the model	59
5.3	Study Design	60
5.3.1	Research Questions	60
5.3.2	Data	61
5.3.3	Baselines	61
5.3.4	Experimental Setup	62
5.4	Results	62
5.4.1	RQ9: Within language code-to-code search	62
5.4.2	RQ10: Cross-language code-to-code search	64
5.4.3	RQ11: StaCE for clone detection	66
5.4.4	RQ12: Exploring Scalability	66
5.4.5	Threats to Validity	69
Chapter 6 Discussion		71
6.1	Dynamic code clone detection	71
6.1.1	Impact of input sizes	72
6.1.2	Types of clones	72
6.1.3	Influence of arguments in clones	73

6.1.4	Clones vs Lines Of Code	73
6.1.5	Extending to support other languages	74
6.1.6	Applications	74
6.2	Cross-language code similarity using non-dominated ranking	76
6.2.1	Scalability and Open-Source Support	76
6.2.2	On the Cost/Benefit of Dynamic Analysis	79
6.2.3	Support for new languages	79
6.2.4	Adding new search similarities	80
6.3	Embedding code similarity	80
6.3.1	Analyzing Results	80
6.3.2	Explaining Similarity	82
6.3.3	Dichotomy of Dynamic Similarity	83
Chapter 7	Related Work	84
7.1	Code Similarity	84
7.1.1	Static Similarity	84
7.1.2	Dynamic Similarity	85
7.2	Code-to-code search	85
7.2.1	Embedding Source Code	86
7.3	Clone Detection	87
Chapter 8	Future Work	88
8.1	Learning Programming Languages using Examples	88
8.2	Performance based Refactoring	89
8.3	Automated Test Suite Generation	90
8.4	Cross-language code search using distributed code representations	90
8.5	Cross-language code similarity for open-source code	91
8.6	TransCompiler	92
Chapter 9	Conclusion	93
BIBLIOGRAPHY		95

LIST OF TABLES

Table 2.1	Types of code similarity. Types I, II and III are static similarity measures while type IV is behavioural similarity [49]	6
Table 3.1	Projects used in this study with the number of valid submissions in both Java and Python.	21
Table 3.2	Number of whole method clones identified by HitoshiIO(H), SLACC(S) and both the approaches, after accounting for false positives.	26
Table 3.3	# of Java, Python and Cross language clusters detected by SLACC compared against AST (Type-III) clusters.	27
Table 4.1	Tokens for functions from Fig. 4.1	35
Table 4.2	Similarity metrics for Java(J) and Python(P) functions from Fig. 4.1. High similarity is associated with high values (\uparrow) of d_{token} , low values (\downarrow) of d_{AST} , and high values (\uparrow) of d_{IO}	36
Table 4.3	Generic ASTs for functions from Fig. 4.1	37
Table 4.4	Summaries of AtCoder and BigCloneBench datasets.	43
Table 4.5	RQ4 & RQ5: Cross-language search results on AtCoder dataset comparing COSAL against the state-of-the-practice ($SotP$) GitHub, and ElasticSearch. COSAL $_{token}$, COSAL $_{AST}$, COSAL $_{SLACC}$ use single search similarities (<i>Single Sim.</i>) d_{token} , d_{AST} and d_{IO} respectively. COSAL $_{static}$ uses d_{token} and d_{AST} with non-domination. KD $_{IO+AST+token}$ performs code search with KDTree using d_{token} , d_{AST} and d_{IO} . Code search techniques using multiple search similarities are represented with <i>Multiple Sim.</i>	48
Table 4.6	RQ6: Single-language Java code search comparing COSAL to the state-of-the-art ($SotA$) FaCoY on AtCoder and BigCloneBench.	49
Table 4.7	RQ6: Performance of FaCoY and GitHub Search compared to COSAL based on BigCloneBench.	50
Table 4.8	RQ7: Performance of COSAL in clone detection compared to ASTLearner, CLCDSA, and SLACC on AtCoder.	51
Table 4.9	Code search results for Haskell on AtCoder comparing COSAL against the state-of-the-practice ($SotP$) GitHub, and ElasticSearch. COSAL $_{token}$, COSAL $_{AST}$, COSAL $_{SLACC}$ use single search similarities (<i>Single Sim.</i>) d_{token} , d_{AST} and d_{IO} respectively. COSAL $_{static}$ uses d_{token} and d_{AST} with non-domination. KD $_{IO+AST+token}$ performs code search with KDTree using d_{token} , d_{AST} and d_{IO} . StaCE performs code search with Embedding using d_{token} and d_{AST} . Code search techniques using multiple similarity measures are represented with <i>Multi Sim.</i>	52

Table 4.10	Cross-language code search results for Haskell queries with Python and Java search results on AtCoder dataset comparing COSAL against the state-of-the-practice (<i>SotP</i>) GitHub, and ElasticSearch. $COSAL_{token}$, $COSAL_{AST}$, $COSAL_{SLACC}$ use single search similarities (<i>Single Sim.</i>) d_{token} , d_{AST} and d_{IO} respectively. $COSAL_{static}$ uses d_{token} and d_{AST} with non-domination. $KD_{IO+AST+token}$ performs code search with KDTree using d_{token} , d_{AST} and d_{IO} . StaCE performs code search with Embedding using d_{token} and d_{AST} . Code search techniques using multiple similarity measures are represented with <i>Multi Sim.</i>	53
Table 5.1	Summaries of AtCoder (AtC) and BigCloneBench (BCB) dataset.	61
Table 5.2	RQ9: Within-language search results in Haskell, Java and Python on AtCoder dataset and Java on BigCloneBench dataset. StaCE is compared against the state-of-the-practice (<i>SotP</i>) GitHub and ElasticSearch and state-of-the-art (<i>SotA</i>) Code2Vec [120] and COSAL. The table reports the MRR (M), Precision@1,3,5,10 (Prec) and SuccessRate@1,3,5,10 (SRate) for the queries and search results in the same language.	63
Table 5.3	RQ10: Cross-language search results on AtCoder dataset in Haskell, Java and Python comparing COSAL and StaCE against the state-of-the-practice (<i>SotP</i>) GitHub and ElasticSearch and state-of-the-art (<i>SotA</i>) Code2Vec [120]. Queries are in each language and search results in the other two languages. $COSAL_{token}$, $COSAL_{AST}$, $COSAL_{SLACC}$ use single search similarities (<i>Single Sim.</i>) d_{token} , d_{AST} and d_{IO} respectively. $COSAL_{static}$ uses d_{token} and d_{AST} with non-domination. $StaCE_{tokens}$ and $StaCE_{AST}$ uses the Tokens Vector and Tree Vector from Fig. 5.3. Techniques using multiple similarity measures are represented with <i>Multi Sim.</i>	64
Table 5.4	RQ11: Cross-language performance of StaCE in clone detection compared to ASTLearner, CLCDSA, SLACC and COSAL on AtCoder for Java and Python. . .	67
Table 5.5	Comparing StaCE and COSAL on varying sizes of {1,2,5,10,20,50,100}% of the AtCoder dataset.	67
Table 5.6	R12: Comparing training and query time in minutes for StaCE and COSAL varying data sizes for {1,2,5,10,20,50,100}% of data. The training time refers to the time taken to build the prediction model and the query time is the time taken to search 1000 random queries.	68
Table 6.1	Mean and variance (in parenthesis) of # clones, # clusters and # false positives for 20 repeats when # inputs varying between 8-256. The mean (and variance) are reported.	72
Table 6.2	Performance of FaCoY and GitHub Search compared to COSAL based on 4,984 code snippets from BigCloneBench [94] benchmark which can be executed. .	79

LIST OF FIGURES

Figure 1.1	Minimum value in array using Java and Python	2
Figure 2.1	Python functions with same tokens but different behavior	6
Figure 2.2	Sample Java and Python functions to group posts based on the post owner. Example and AST from [125].	7
Figure 2.3	General architecture of embedding objects	10
Figure 3.1	High level workflow for SLACC.	14
Figure 3.2	An example depicting conversion of a function with object as a return type to multiple functions with non-primitive members of the object's class.	16
Figure 3.3	An example illustrating the need for reordering arguments. The two func- tions perform integer division but do not return the same return value for the same set of inputs due to the order of arguments in the function definition.	17
Figure 3.4	Semantic clusters detected by HitoshiIO, SLACC on method level (SLACC _{method}) and SLACC on statement level (SLACC _{stmt}). The cluster contains functions that take an object that reads a file and returns the next Integer token.	25
Figure 3.5	Semantic cluster of Python functions detected by SLACC. The cluster con- tains functions that returns the sum of an input array.	27
Figure 3.6	Semantic cluster of a Java function and a Python function detected by SLACC. The cluster contains functions that returns the minimum value in an input integer array.	28
Figure 4.1	Different functions to return a filtered array of numbers implemented in Java and Python. The code in (a), (d), and (e) are functionally identical.	32
Figure 4.2	Overview of COSAL	34
Figure 4.3	Normalized AST for the Java statement <code>'int x = y + 15 * 2.0;'</code>	38
Figure 5.1	Comparing runtimes for 100 random queries comparing Code2Vec and COSAL	55
Figure 5.2	Java and Python implementations to get the n^{th} value from the Fibonacci series.	55
Figure 5.3	Overview of StaCE	56
Figure 6.1	Cumulative # clones with # arguments varying between 1-5.	73
Figure 6.2	# clones for lines of code between ranging from 1-29. Clones with 30 or more lines are grouped into 30+	74
Figure 6.3	Functions from Open-Source Java library from commons by apache and guava by google and Python library collections	77
Figure 6.3	Functions from Open-Source Java library from commons by apache and guava by google and Python library collections (cont.)	78
Figure 6.4	Snippets from four valid solutions for the connectivity problem from AtCoder. All four snippets show implementations the Union-Find data structure.	81
Figure 6.5	Percentage of overlapping results for StaCE vs COSAL for {1,2,5,10,20,50,100}% of data for 100 random queries in Haskell, Java and Python.	82

CHAPTER

1

INTRODUCTION

Modern programmers typically work on systems built with a cocktail of multiple programming languages [42]. A recent survey found that professional software developers have a mean of seven different programming languages in their industrial software projects [107] and open-source software projects frequently have between 2–5 programming languages [86, 90]. To learn a new programming language, studies have shown that programmers attempt to use a *cross-language* learning strategy by reusing knowledge from a previously known language [9, 10, 117]. This means programmers often need the ability to relate code snippets across multiple programming languages. However, finding similar code across languages is not limited to learning new programming languages. It is a precursor to multiple applications in software engineering ranging from duplicate code detection [125, 127], finding translations of code in a different language [108] and automated program repair [48, 80]

Code similarity is the task of identifying similar snippets of code across different programming languages (cross-language) or within a programming language (in-language) using a specific similarity measure. Code similarity can be broadly categorized as static or dynamic based on the representation of code used to measure the similarity. Most code similarity techniques compare code within a programming language [14, 18, 26, 30, 47, 77, 115]. Some techniques [47, 102, 120] though intended for in-language code similarity can be extended to support cross-language code similarity. A handful of techniques are explicitly designed for cross-language code similarity but limited to languages that are similar to each other like Java and C# [108, 125]. However, none of these techniques can be used to find similar code across languages that belong to different programming paradigms like Java and Haskell or with dynamic type systems like Python.

Traditional approaches for comparing source code, both in-language and cross-language, primarily rely on static analysis techniques like comparison of tokens [14, 26, 30] and abstract syntax

trees [18, 77, 125]. The success of such measures is heavily predicated on “how the developer codes” rather than “how the code behaves”. For example, Fig. 1.1 shows a Java and Python implementation for finding the minimum value in an array. The Java function `min_java` iterates over the array and returns the minimum while the Python function `min_py` uses an external function `reduce`. A static analysis approach would fail to detect that these functions are behaviorally similar since they use different tokens and have different ASTs. However, a dynamic analysis approach based on comparing the Input-Output would identify that these functions produce same outputs for the same inputs.

```
1 int min_java(int[] arr) {
2     int min = arr[0];
3     int l = arr.length
4     for (int i = 1; i < l; i++)
5         if (arr[i] < min)
6             min = arr[i];
7     return min;
8 }
```

```
1 def min_py(lst):
2     import functools
3     return functools.reduce(
4         lambda a,b : \
5             a if a < b else b,
6         lst
7     )
```

Figure 1.1 Minimum value in array using Java and Python

Some techniques use dynamic approaches for code similarity like comparing Input-Output (IO) relations [47, 102] or instruction graphs. These techniques suffer from limitations like different typing schemes for languages, limitations on profiling of code, low recall and limited scalability [56]. Hence, cross-language code similarity as a result, until recently has been computed based on static attributes. However, dynamic code similarity approaches has its merits like high precision and high interpretability [47]. However, there exists a lack of demonstrable cross-language code similarity tool using dynamic similarity and how limitations of a similarity measure can be overcome by using it in tandem with other similarity measures. This work advances the knowledge of the thesis statement:

Combining static and dynamic code similarity metrics impacts precision, recall, and scalability in code-to-code search and clone detection applications.

1.1 Overview

This research aims at studying and developing similarity measures to compare different programming languages. The similarity measures developed are compared in the context of code clone detection and cross-language code-to-code search.

The research begins with SLACC, a cross-language code clone detection tool. SLACC demonstrates the feasibility and the merits of using a dynamic similarity measure to compare code across programming languages. However, SLACC suffers from low recall, does not scale to open-source code and is predicated on the execution of source code. The second tool COSAL overcomes some of these limitations by augmenting SLACC with token-based and AST-based static similarity measures

using non-dominated ranking. The feasibility of COSAL is demonstrated using cross-language code-to-code search across Java, Python and Haskell. Finally, the last tool StaCE generates embeddings based on the static similarity measures from COSAL. StaCE and COSAL have similar performance metrics but have trade-offs based on the applications. The merits and demerits of COSAL and StaCE is explored at the end of this research.

1.2 Contributions

Each tool developed as part of this research is published at ICSE 2020 [134], FSE 2021 [139], submitted to PLDI 2022 [140] and awaiting a journal submission. Here are the contributions of this study:

- An empirical validation for single-language static typed clone detection demonstrating SLACC is more precise (86.7% *vs.* 30.7%) than the state-of-the-art code-clone detection technique HitoshiIO (section 3.9.1).
- The first exploration of cross-language clone detection when the languages lack an underlying representation; SLACC is successful in identifying cross-language clone clusters between Python and Java with 94.1% precision (section 3.9.2, section 3.9.3).
- COSAL, a first code-to-code search using multi-objective search over static and dynamic search objectives (Chapter 4),
- A comprehensive evaluation of COSAL with state-of-the-practice code search techniques in GitHub and Elasticsearch (section 4.9.2), with state-of-the-art code search techniques FaCoY (section 4.9.3) and cross-language clone detection technique CLCDSA and SLACC (section 4.9.4).
- A demonstration of COSAL across languages with different programming paradigms, functional programming (Haskell) and object-oriented programming (Java).(section 4.9.4)
- StaCE, a first cross-language code-to-code search by embedding multiple search similarities (Chapter 5).
- An evaluation of StaCE against state-of-the-practice and state-of-the-art cross-language code-to-code techniques (section 5.4.1, section 5.4.2) and state-of-the-art code-clone detection tools (section 5.4.2).
- Open source tool for SLACC, COSAL and StaCE. ¹

1.3 Outline

The rest of this thesis document begins with a background on source code similarity and its different associated components in Chapter 2. Next, Chapter 3 presents SLACC and cross-language

¹github.com/dr-bigfatnoob/CodeSeer

code clone detection using dynamic similarity. Chapter 4 presents COSAL and cross-language code-to-code search using static and dynamic search similarities. Chapter 5 presents StaCE and embedding multiple search similarities for cross-language code similarity. In Chapter 6 we discuss the implications, limitations and a discussion on the results of all three tools. Chapter 7 presents the related literature to this research. Finally, a broader look at future work for this study is presented in Chapter 8 followed by a conclusion in Chapter 9.

CHAPTER

2

BACKGROUND

This chapter encompasses basics of source code similarity, multi-objective code search and embedding objects used in this work.

2.1 Similarity of Code

Source code similarity is used to characterize the relationship between snippets of code in software engineering applications such as program repair [53, 63, 74, 85, 89], code search [115, 123, 127], software security [40, 77, 119] and identifying plagiarized code [14]. Code similarity can be measured in a variety of ways, including textually, structurally, or semantically. Roy et al. [49] categorizes similar code snippets into four types based on the differences between the snippets.

In the literature, Types I, II and III are classified based on syntactic measures of similarity (i.e how the code looks) and Type IV clones are based on semantic measures of similarity (i.e., how the code behaves).

However there is an interplay between this broad classification of syntax and semantics particularly when it comes to Type IV clones or functionally similar code fragments. Kim et al. [115] use tokens extracted from source code to determine semantically similar code. On the other hand, Su et al. [102] uses dynamic information, specifically an Input-Output based profiling approach, in determining Type IV code clones. On the code search front, S6 [48] uses a hybrid approach involving both syntactic and dynamic Input-Output (IO) behavior to identify functional code clones. It is unclear based on the literature if semantic code similarity is based on static information, dynamic information, or a combination. As a result, to avoid such ambiguities, we classify code similarity measures and techniques based on the analyses used: static or dynamic.

Table 2.1 Types of code similarity. Types I, II and III are static similarity measures while type IV is behavioural similarity [49]

Type	Description
I	Identical sans whitespace and comments
II	Identical AST but uses different variable names, types or function calls
III	Similar AST but uses different expressions/statements. For example, a) using <code>while</code> in place of <code>for</code> loops or b) using <code>if else if</code> in place of <code>switch</code> statements.
IV	Different syntax but behaviorally same. For example, an iterative stack approach or a recursive approach can be used for breadth first search of a graph.

```

1 def f1(x):
2     if x > 5:
3         print "more"
4     if x < 10:
5         print "less"

```

```

1 def f2(x):
2     if x > 5:
3         print "more"
4     if x < 10:
5         print "less"

```

Figure 2.1 Python functions with same tokens but different behavior

2.1.1 Static Code Similarity

Techniques that use static code attributes to compute similarity often parse code into an intermediate representation and then compare the intermediate representations to compute a measure of syntactic similarity. Based on the choice of intermediate representations, such techniques can be classified into text-based [14, 26, 30], AST-based [18, 38] and graph-based [43, 61].

2.1.1.1 Text based Approaches

Similar code within and between languages tend to use the similar vocabulary. Similarity studies show that chunks of code within the same language and that perform the same task tend to use the same tokens [31]. This is the prime intuition behind text based approaches [26, 30, 110, 130]. Libraries across languages tend to share similar names based on their functionalities; for example `List` class from `java.util` library and `list` from the built-in `Python` `system` library is both commonly used to represent an array. Similarly, open source projects follow a code style of naming variables based on the functionality [78].

2.1.1.2 Tree based Approaches

Tree based similarity approaches compare the similarity between these tree based representation of code as a proxy for the similarity between code [18, 29, 38, 125]. In most cases [18, 38, 125] the code is represented as the Abstract Syntax Trees (AST) and subsequently compared. Consider the example

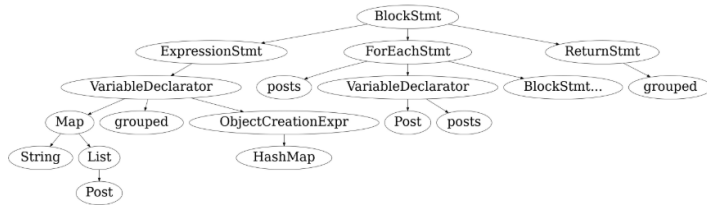
of grouping list of posts in Java and Python as shown in Fig. 2.2a and Fig. 2.2c. The corresponding ASTs Fig. 2.2b and Fig. 2.2d can be seen to show similarities although they are programmed in different programming languages and encoded as ASTs using different libraries. For example, both the body of the Python function `group_post` and the Java method `groupPost` contain three children – an assignment, a `for` loop and a `return` statement.

```

1  Map groupPosts(List<Post> posts) {
2    Map<String, List<Post>> grouped =
3      new HashMap<>();
4    for (Post post: posts) {
5      if (!grouped.containsKey(post.getOwner()))
6        grouped.put(post.getOwner(),
7          new ArrayList<Post>());
8      grouped.get(post.getOwner()).add(post);
9    }
10   return grouped;
11 }

```

(a) Java: `groupPosts` method in Java



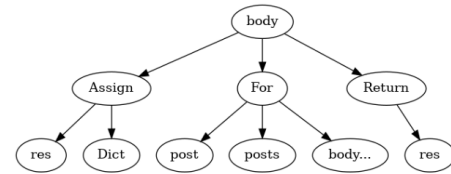
(b) AST for `groupPosts` method in Java using JavaParser [87].

```

1  def group_posts(posts):
2    res = {}
3    for post in posts:
4      b = res.setdefault(
5        post.owner, [])
6      b.append(posts)
7    return res

```

(c) Python: `group_posts` method in Python



(d) AST for `group_posts` method in Java using Python-AST module [152].

Figure 2.2 Sample Java and Python functions to group posts based on the post owner. Example and AST from [125].

Although we can see similarities between the ASTs, due to different notations and lack of a uniform AST encoding module across different languages, most similarity techniques function only within the scope of a single programming language [18, 38]. In some cases, the trees are projected onto a uniform vector space and comparisons are made on this space instead. Perez and Chiba use this approach where they succeed in comparing cross language clones by using an unsupervised learning approach to learn vector representations from the ASTs and an LSTM based neural network to measure the similarity between the vector representations [125].

Later in this study (section 4.2.2) we propose a common grammar for AST using similar notations to encode code from Java and Python and use node edit distance between the trees as a measure of syntactic similarity.

2.1.1.3 Graph based Approaches

In graph based similarity approaches code using a graphical representation and similarity is measured as a proxy of the difference between the graphs. In most of these approaches [21, 22, 36, 43],

code is generally represented as Program dependence graphs [7] (PDGs) which is a representation, using graph notation, that makes data dependencies and control dependencies explicit. Komondoor and Horwitz [21] apply program slicing on PDGs generated from C programs. Krinke et al. also detect isomorphic subgraphs with maximum size k after generating PDGs. In this case, the PDGs are constructed at a much finer granularity where each vertex roughly maps to a node in the AST. Gabel et al. [43] uses a combination of AST and PDG. They first generate the PDF of a method and maps it to an AST. They then compare similarity of the AST using Deckard [38].

When it comes to cross-language syntactic similarity, most techniques are text-based [108, 124] or a combination of tree-based and text-based [44, 125]. A major limitation of text-based approaches is in ignoring syntactic constructs. For example, consider functions `f1` and `f2` from Fig. 2.1, which have the same set of tokens extracted but are structurally different since Python uses indentation to represent code block levels and nesting. Tree- and graph-based approaches have not been explored for cross-language similarity. We suspect this can be attributed to different annotations in grammars used by standard language parsers like ANTLR [76], JavaParser [87] and `python-ast` [152]. Later in the study, we overcome this challenge by using a common language-agnostic grammar based on abstracting out common features across programming languages to build a generic AST (section 4.2.2).

2.1.2 Dynamic Code Similarity

Techniques that execute code to determine similarity are classified as dynamic. These measures typically rely on identifying input parameters and then monitoring the behavior of the code as it runs. For some techniques, functions are adjudged to be similar if they have similar inputs, outputs and side-effects [47, 58, 102, 134]. Other techniques use abstract program states after executions to analyze the behaviors of the code fragments [54, 101, 126]. Dynamic measures are particularly successful in detecting code-clones across programming languages since it does not rely on syntactic properties [47, 134]. These approaches include multiple limitations such as

- **Execution:** Since the code fragments need to be dynamically profiled, they need to be valid executable fragments of code. This leads to additional challenges in dependency management and syntactic validation. As a result the number of valid standalone executable code snippets can be limited.
- **Granularity:** Such techniques work mostly on a method level granularity. Functioning at lower level granulates can lead to memory explosion in larger repositories [56]
- **Runtime:** Finally, such techniques can have a large runtime since the methods need to be profiled for a large number of inputs.

2.2 Pareto Dominance

A **Multi-Objective Search (MOS)** is a category of mathematical search problem which involves more than one objective function to be searched across simultaneously. Mathematically, a MOS can be defined as

$$\min_{x \in \Omega} F(x) = (f_1(x), \dots, f_m(x)) \quad (2.1)$$

where Ω is the *decision (variable) space*, R^m is the objective space and $F : \Omega \rightarrow R^m$ consists of m real-valued objective functions. If Ω is a closed and connected region in R^n and all the objectives are continuous of x , the problem in Eq. 2.1 is categorized as a *Continuous Multi-Objective Search Problem*. Note that Eq. 2.1 represents a minimization problem where an objective for two points is considered to be better/dominate if it has a *smaller* value. It can also be applied for a maximization problem where a *larger* value is considered as better. All the inequalities is reversed to maximize the objective. For consistency and simplicity, we consider *better* to be *smaller* in this section.

Classically, search methods [5, 15] and evolutionary algorithms [13] convert multi-objective search to a single-objective search problem [62]. For example, linear scalarization is a common approach taken to convert a multi objective problem to a single objective one by using a weighted sum of all the objectives $\min_{x \in \Omega} \sum_{i=1}^m w_i * f_i(x)$. This approach would require additional challenges in tuning the weights (w_i) and a loss of information of the individual objectives. Using such methods is not very optimal as ranking the results would be very subjective if the objectives are independent (which ours mostly are).

Hence, we rank two search results using Pareto Dominance [62]. For two

Let $a = (a_1, \dots, a_m), b = (b_1, \dots, b_m) \in R^m$ be two objective vectors, then a is said to *dominate* b if $a_i \leq b_i$ for all $i = 1, \dots, m$, and $a \neq b$.¹ A point $x^* \in \Omega$ is called (*globally*) *Pareto² optimal* if there is no $x \in \Omega$ such that $F(x)$ dominates $F(x^*)$. The set of all the Pareto optimal points is called the *Pareto Set (PS)*. The set of all the Pareto objective vectors is defined as $PF = \{F(x) \in R^m | x \in PS\}$, is called the *Pareto Front*[62].

2.3 Embedding Objects

Object embedding is a means to represent an object as a vector by projecting it onto lower dimensions such that similar objects are closer when compared using the embeddings. Fig. 2.3 represents the general architecture of an object embedding model. The figure contains three major components: an *object representation layer*, an *embedding layer* and the embedded representations (*Embeddings*). Consider color as a feature we want to use to adjudge similarity. Objects A and B share a similar shade of color, as do objects C and D. Ideally, the objects should be embedded such that distance between the embeddings of the pairs (A,C) and (B,D) are minimum.

¹This is described for a minimization problem. All the inequalities is reversed to maximize the objective in [62]. "Dominate" refers to "better than".

²Named after Vilfredo Pareto, an Italian engineer and economist

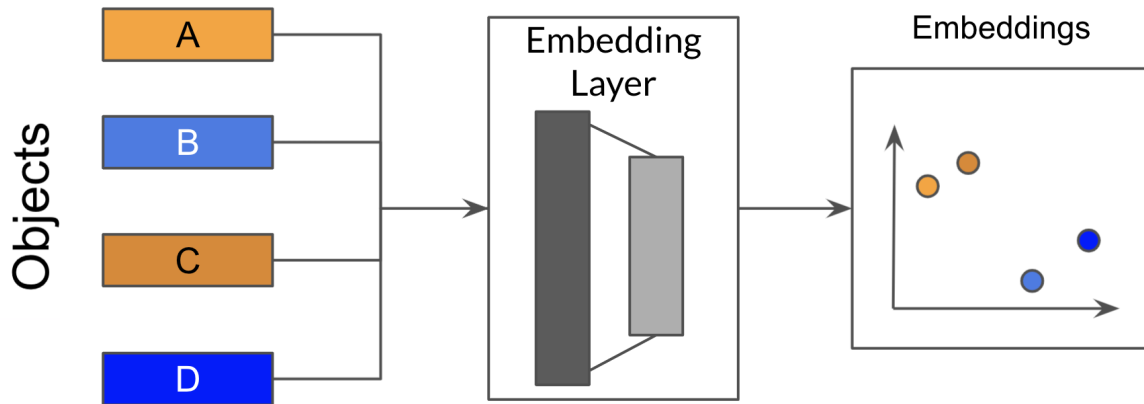


Figure 2.3 General architecture of embedding objects

The representation of the object varies based on the nature of the object and similarity relationship. For example, textual objects are represented using one-hot encoding where each word is represented as sparse vector with one components set to 1 and all other components set to 0. These encoded representations are sent as input to the embedding layer based on the relationship between the individual objects. For example, if the sequential order of words influences the relationship of the objects (like a paragraph), a Continuous-Bag-Of-Words (CBOW) approach can be used to input the objects. In contrast, if the order of the objects has no influence on the relationship, a Term-Frequency-Inverse-Document-Frequency (TFIDF) approach can be adopted. Similar approaches can be used for encoding images [39], speech [112] and in our study even code [114, 120].

The choice of the embedding layer depends on the nature of the dataset, similarity relationship and computing resources. For textual data, a popular approach is the skip-gram model [34] where the model model is trained to predict a word based on its neighboring words. Similarly, for longer sequential data an LSTM [97] based approach can be used to generate embeddings. With the rise of computation power, attention based methods have gained popularity [91].

CHAPTER

3

SIMION BASED LANGUAGE AGNOSTIC CODE CLONES

This work was published in ACM International Conference of Software Engineering (ICSE) 2020 in collaboration with Dr. Chris Parnin and Dr. Kathryn T Stolee [134].

3.1 Motivation

Avery is preparing for a technical interview and was given a few practice coding challenges [129] to work on. Avery is more comfortable writing code in Java during an interview setting but is worried because the company exclusively codes in Python. As practice for the interview, Avery wants to code with Python. First, Avery decides to write the code in Java to understand the solution, and then translate those solutions into Python code.

One of the practice questions asks the coder to interleave the results of two arrays. Avery quickly writes this solution in Java:

```

1 public String interleave(int[] a, int[] b) {
2     String result = "";
3     int i = 0;
4     for( i = 0; i < a.length && i < b.length; i++ ) {
5         result += a[i];
6         result += b[i];
7     }
8     int[] remaining = a.length < b.length ? b : a;
9     for( int j = i; j < remaining.length; j++ ) {
10        result += remaining[j];
11    }
12    return result;
13 }

```

While one approach is to directly translate the code into Python, Avery wonders if there are other ways to take advantage of idioms and capabilities in Python. After spending a few hours searching Stack Overflow [157] and GitHub Gists [154], Avery finds a few code snippets that seem to do the same thing.

The first one seems a bit too complex and relies on another dependency.

```

1 def fancy_interleave(l1, l2):
2     from itertools import chain
3     return "".join([str(x)
4                     for x in chain.from_iterable(zip(l1, l2))])

```

This other solution is similar to the Java solution, but is using something new, a zip function. Avery is excited to learn some new Python tricks!

```

1 def problem2(l1, l2):
2     result = ""
3     for (e1, e2) in zip(l1, l2):
4         result += str(e1)
5         result += str(e2)
6     return result

```

Avery found the strategy of writing code in Java and translating that code into Python helpful. However, the process of manually searching and translating the code between languages was time-consuming. Avery's unfamiliarity with Python made it difficult to verify whether these snippets were *truly* the same.

At the interview, Avery was relieved to be asked to solve the same *interleave* problem from the practice set! However, while coding up a solution in Python, the interviewer asked, *does this handle interleaving uneven lists?* The original Java-based solution handled this case, but the Python translation did not. Because searching for code took so long, Avery never had the chance to fully verify that the Python solution worked the same as the Java solution. Avery's assumption that the

new zip function would work on uneven lists was wrong! Had there been a better way for Avery to find semantically related snippets in other programming languages, this issue may have been avoided.

This work introduces COSAL, which could detect that these functions are not equivalent. From a corpus of code, it could instead find this semantically identical snippet—just one of many applications enabled by cross-language clone detection:

```
1 def valid_interleave1(l1, l2):
2     result = ""
3     a1, a2 = len(l1), len(l2)
4     for i in range(max(a1, a2)):
5         if i < a1:
6             result += str(list1[i])
7         if i < a2:
8             result += str(list2[i])
9     return result
```

3.2 SLACC

Code clones can be broadly classified into four types [49] as described in Table 2.1. Types I, II and III represent static code clones where similarity between code is estimated with respect to the structure of the code. On the other hand, type-IV indicates functional similarity. Static code clone detection techniques are impractical for cross-language code clone detection as it would require an explicit mapping between the syntax of the languages. This is feasible for syntactically similar languages like Java and C# [144] but much harder for different languages like Java and Python. On the other hand dynamic approaches for cross-language code detection [108] rely on large number of training examples between the languages and was yet again tested on similar programming languages.

Code-to-Code Search Across Languages (SLACC) is a semantic approach to code similarity that is predicated on the availability of large repositories of redundant code [80]. Instead of mapping API translations using predefined rules [4, 144], or using embedded API translations [23, 108], SLACC uses IO examples to cluster code based on its behavior. Further, it relaxes the bounds of the datatypes across programming languages, which helps dynamic typed code snippets (e.g., Python) to be clustered alongside static typed code snippets (e.g., Java).

SLACC builds on the ideas pioneered by EQMiner [47] for using segmentation and random testing for clone detection. SLACC starts by identifying snippets from a large code base and involves a multi-step process depicted in Fig. 3.1, which starts with a) *Segmentation* of the code base into smaller fragments of code called snippets, b) *Function creation* from the snippets, c) *Input generation* for the functions, d) *Execution* of the functions, and e) *Clone detection* based on clustering functions arguments and execution results.

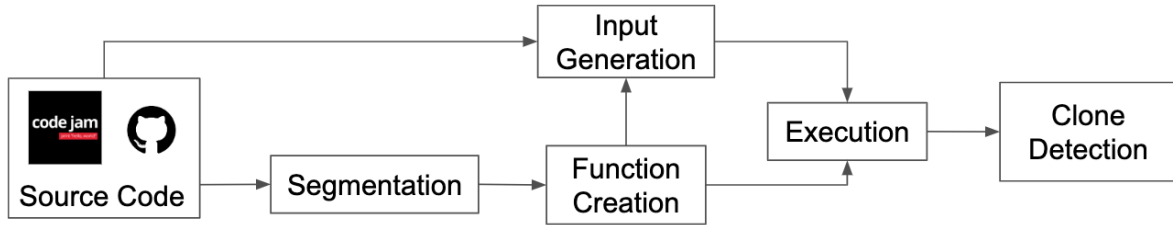


Figure 3.1 High level workflow for SLACC.

3.2.1 Segmentation

In the first stage, code from all the source files in a project is broken into smaller code fragments called *snippets*. Consecutive statement blocks of threshold `MIN_STMT` or more are grouped into a snippet. A statement block can be

1. **Declaration** Statement. *e.g.*, `int x;`
2. **Assignment** Statement *e.g.*, `x = 5;`
3. **Block** Statement *e.g.*, `static {x = 10;}`
4. **Loop** statements. *e.g.*, `for`, `while`, `do-while`
5. **Conditional** statements. *e.g.*, `if`, `if-else-if`, `switch`,
6. **Try** Statement. *e.g.*, `try`, `try-catch`

Algorithm 1 illustrates the segmentation phase. For an AST A_F of a function, the algorithm performs a pre-order traversal of all the nodes in the AST (*line 5*) and then uses a sliding window to extract segments of size greater than a minimum segment size `MIN_STMT` (*lines 12-13*). Further, for statements like Block, Loop, Conditional and Try which have statements in its nested scope, the algorithm is called recursively on them (*lines 14-15*).

3.2.2 Function Creation

Next, snippets are converted into executable functions. This section describes how arguments, return variables, and types are inferred.

3.2.2.1 Inferring arguments and return variables

SLACC adapts a dataflow analysis similar to that used by Su et al. [102]. For each method, potential return variables are identified as variables that are defined or modified within the scope of the snippet. If the last definition of a variable is a constant value, that variable is removed from the set of potential return variables. Arguments are variables that are 1) used but not defined within the scope of the snippet, and 2) not declared as public static variables for the class. For each variable,

Algorithm 1 Segmentation

```
1: Input:  $A_F$  - AST Node
2: Output:  $\mathbb{S}$  - List of Segment
3: procedure SEGMENT( $A_F$ )
4:    $\mathbb{S} \leftarrow \phi$ 
5:    $stmts \leftarrow PreorderTraverse(A_F)$ 
6:   for all  $i \in \text{range}(0, \text{len}(stmts) - 1)$  do
7:      $S_i \leftarrow \{\}$ 
8:      $stmt_i \leftarrow stmts[i]$ 
9:     for all  $j \in \text{range}(i, \text{len}(stmts))$  do
10:       $stmt_j \leftarrow stmts[j]$ 
11:       $S_i.append(stmt_j)$ 
12:      if  $\text{len}(S_i) \geq MIN\_STMTS$  then
13:         $\mathbb{S} \leftarrow \mathbb{S} \cup S_i$ 
14:      if  $stmt_j.hasChildren()$  then
15:         $\mathbb{S} \leftarrow \mathbb{S} \cup \text{SEGMENT}(stmt_j)$ 
16:   return  $\mathbb{S}$ 
```

the scope and positions of declaration, initialization, usages and modifications are recorded. Using this metadata, for a snippet, variables that have not been defined at the start of the snippet and variables that have been modified within the snippet are identified. The undefined variables become arguments of the function while the modified variables become potential return variables. For each potential return variable in a snippet, a function is created.

3.2.2.2 Inferring types

In the case of static typed languages, argument types and return values can be inferred via static code analysis. For dynamic typed languages, the parameters can take multiple types of input arguments. This increases the possible values of the arguments generated (see section 3.2.3) to identify its behavior. In many cases, the possible types for the arguments can be inferred by parsing the code and looking for constant variables [3] in its context. This technique has been used in inferring types in other dynamic languages like JavaScript [46]. For example, in the following Python function, the type of n can be assumed to be an integer since it is compared against an integer.

```
1 def fib(n):
2     if n <= 1: return n
3     return fib(n-1) + fib(n-2)
```

In cases where the types of the parameters could not be inferred at compile time, such as:

```
1 def main(a):
2     print a
```

a generic type is assigned (i.e., for a) allowing the argument to assume any of the primitive types used in argument generation (section 3.2.3).

```

1  class Shape {
2      public int length;
3      int width;
4      private int height;
5      public Shape(int l, int w, int h) {
6          length=l; width=w; height=h;
7      }
8  }
9  public Shape func_s (int l, int w, int x) {
10     return new Shape(l + x, w * 2, x);
11 }
12 public int func_l (int l, int w, int x) {
13     return func_s(l, w, x).length;
14 }
15 public int func_w (int l, int w, int x) {
16     return func_s(l, w, x).width;
17 }

```

Figure 3.2 An example depicting conversion of a function with object as a return type to multiple functions with non-primitive members of the object's class.

3.2.2.3 Converting object return types into functions

If a snippet returns an object, the object is simplified into multiple functions returning each of its non-private members independently. For example, in Fig. 3.2, `func_s` has a return type of `Shape`. `Shape` has two members, `length` and `width`. Hence, `func_s` is broken down into two functions, `func_l` and `func_w`, which return the `length` and `width` of the shape object independently. Note that a third function for `height` is not created since it is a private member.

3.2.2.4 Permuting argument order

For each of the snippets, different permutations based on the input of arguments are generated since order matters for capturing function behavior. Consider the two functions in Fig. 3.3; the first function divides `a` with `b` using the division (`/`) operator while the second divides `dividend` with `divisor` using the subtract (`-`) operator recursively. For the inputs (5, 2) the two functions would produce the values 2 and 0 respectively. But if the arguments for the second function was reversed, it would produce the same output 2. Thus, for every function, duplicates in different permutations of the arguments (ARGS) are created, resulting in $|\text{ARGS}|!$ different functions. To limit the creation of this exploding space, an upper limit on the number of arguments per function is set that is included in the analysis (ARGS_MAX).

3.2.3 Input Generation

A set of inputs are required to execute the created functions. Following this, clustering is performed.

```

1 public int divide_simple (int a, int b) {
2     if (b == 0) return 0
3     return a / b;
4 }
5 public int divide_complex (int divisor, int dividend) {
6     // Same as dividend/divisor
7     if (b == 0) return 0
8     int quotient = 0;
9     while (dividend >= divisor) {
10        dividend = dividend - divisor;
11        quotient++;
12    }
13    return quotient;
14 }

```

Figure 3.3 An example illustrating the need for reordering arguments. The two functions perform integer division but do not return the same return value for the same set of inputs due to the order of arguments in the function definition.

3.2.3.1 Input creation

Inputs are generated based on argument type and using a custom input generator inspired by grey-box testing [60] and multi-modal distribution [47]. First, the source code is parsed and constants of each type are identified. Next, a multi-modal distribution is declared for each of the types with peaks at the constants. Finally, values for each type are sampled from this multi-modal distribution. Our experiments create 256 inputs per function, as justified in section 6.1.1.

3.2.3.2 Memoization

For every function with the same argument types, a common set of inputs have to be used to compare them. This is ensured using a database and the input generator. The generator is used to create sample inputs for the given argument types and stored in the database. For subsequent functions with the same signature for the arguments, the stored input values are reused.

3.2.3.3 Supported argument types

SLACC currently supports four types of arguments.

1. **Primitive.** The multi-modal distribution for the argument type is sampled to generate the inputs. This includes integers (and longs, shorts), floats (and double), characters, booleans, and strings.
2. **Objects.** Objects are recursively expanded to their constructor with primitive types; inputs are generated for the types.

3. **Arrays.** A random array size is generated using the input generator for integers¹. For each element in the array, a value is generated based on the array type (Primitive or Object).
4. **Files:** Files are stored as a shared resource pool of strings in the database. If a seed file(s) is provided, it is randomly mutated and stored as a string in the database. In the absence of a seed, constants from the multi-modal distribution are sampled and stored as strings. For an argument with a `File` type (or its extensions), a temporary (deleted on termination) file object is created using the stored strings.

3.2.3.4 Type size restrictions

Comparing code snippets requires compatible sizes of types across programming languages. For example, Java has 4 integer datatypes `byte`, `short`, `int` and `long` which occupy sizes of 1, 2, 4 and 8 bytes, respectively. On the other hand, Python has two integer datatypes: `int` which is equivalent to the `long` datatype in Java and `long` which has an unlimited length. Thus, a restriction is made when generating inputs for functions across different languages: inputs are generated from the smaller bound of the two programming languages. For example, in the case of Java and Python function that has an `int`, inputs are generated within the bounds of Java.

3.2.4 Execution

In the next stage, the created functions are executed over the generated input sets and the subsequent return values are stored. Each function is assigned an execution time limit of T_L seconds, after which a Timeout Exception is raised. This occurs most frequently when there is an infinite loop, such as `while(true)` when the loop invariant is an argument. Each execution of the function is run on an independent thread. Subsequently, the return value, runtime and exception for the executed function over the input set is stored.

3.2.5 Clone Detection

The last stage of SLACC is identifying the clones, where the executed functions are clustered on their inputs and outputs. SLACC uses a *representative based partitioning strategy* [49, 102] to cluster the executed functions.

3.2.5.1 Similarity Measure

In this work, a pair of functions have the highest semantically similarity if for any given input, the functions return the same output. The similarity measure between two functions is computed as the number of inputs for which the methods return the same output value divided by the number of inputs, same as the Jaccard index. This creates a similarity value between two functions with a range of [0.0, 1.0] with 1.0 being the highest.

¹If a negative integer is sampled, the distribution is re-sampled.

Algorithm 2 Clustering

```
1: Input:  $\mathbb{F}$  - List of Functions with Input and Output
2: Output:  $\mathbb{C}$  - List of clusters
3: procedure CLUSTER( $\mathbb{F}$ )
4:    $\mathbb{C} \leftarrow \phi$ 
5:   for all  $F \in \mathbb{F}$  do
6:     for all  $C \in \mathbb{C}$  do
7:        $O \leftarrow \text{GetRepresentative}(C)$ 
8:       if  $\text{Similarity}(O, F) \geq \text{SIM\_T}$  then
9:          $C \leftarrow C \cup F$ 
10:      break
11:   if  $\forall C \in \mathbb{C}, F \notin C$  then
12:      $C_{|\mathbb{C}|+1} \leftarrow F$ 
13:     SetRepresentative( $C_{|\mathbb{C}|+1}, F$ )
14:      $\mathbb{C} \leftarrow \mathbb{C} \cup C_{|\mathbb{C}|+1}$ 
15:   return  $\mathbb{C}$ 
```

Consider the functions from section 3.1, `interleave`, `fancy_interleave`, and `valid_interleave`. For values $a = [2, 3]$ and $b = [4]$, we see that `interleave(a, b) = [2, 4, 3]`, `fancy_interleave(a, b) = [2, 4]` and `valid_interleave(a, b) = [2, 4, 3]`. Functions `interleave` and `valid_interleave` are similar since they have the same output for the same input but `interleave` and `fancy_interleave` are not similar. In contrast, for $a = [2, 3]$ and $b = [4, 5]$, all three functions would return the same output `[2, 4, 3, 5]`. Based on these two inputs, `interleave` and `fancy_interleave` have a similarity of 0.5, `interleave` and `valid_interleave` have a similarity of 1.0, and `fancy_interleave` and `valid_interleave` have a similarity of 0.5. This process is repeated for many such inputs a and b to compute similarity scores between each pair of functions.

Functions are only compared if they have the same number of arguments and cast-able argument types. For example, consider the four functions `f1(int a, String b)`, `f2(long a, File b)`, `f3(File a, String b)` and `f4(String a)`. Functions `f1` and `f2` can be compared since `int` can be cast to a `long` value. But they cannot be compared to `f3` since primitive types cannot be cast to `File`. Similarly, `f1`, `f2` and `f3` cannot be compared `f4` due to the difference in number of arguments.

3.2.5.2 Clustering

A function is compared to a cluster by measuring its similarity with the first function added to the cluster (called *representative*). The clustering algorithm is briefly described in Algorithm 2. An empty set of clusters is first initialized (*line 4*). Each function (*line 5*) is compared against each cluster (*line 6*). If the similarity between the *representative* (*line 7*) and the function is greater than a predefined similarity threshold, `SIM_T` (*line 8*), the function is added to the cluster (*line 9*). If the function does not belong in any cluster (*line 11*), a singleton cluster is created for the function (*line 12*) and the function is set as the cluster's *representative* (*line 13*). The singleton cluster is added to the set of clusters (*line 14*)

3.3 Research Questions

Our goal is to evaluate the effectiveness of SLACC. There is a three-phase evaluation, first to compare SLACC to a comparable technique in a single, static typed language. Next, SLACC is applied to a single, dynamic typed language (Python) and then to a multi-language context; in both cases SLACC is compared to type-III clones.

SLACC is benchmarked against HitoshiIO [102] with respect to coverage and precision of code-clone detection. This leads us to our first research question:

Research Question 1

How effective is SLACC on semantic clone detection in static typed languages?

Prior research has already shown that semantic clones can be found in static typed languages [47, 57, 102] like C and Java. The literature search for this study failed to find techniques that identified semantic code clones in dynamic typed languages. Therefore, an AST based comparison approach is used as an alternative baseline to benchmark SLACC. This leads to the next research question:

Research Question 2

How effective is SLACC on semantic clone detection in dynamic typed languages?

Prior work identified code clones between languages by mapping APIs between similar languages (e.g., Java and C#) using predefined rules [144] or using an embedded API translations [23, 108]. As a result, these code clones are syntactic rather than semantic. Therefore:

Research Question 3

How effective is SLACC at cross-language semantic clone detection?

3.4 Data

SLACC is validated on four problems from Google Code Jam (GCJ) repository and their valid submissions in Java and Python. GCJ is an annual online coding competition hosted by Google where participants solve the programming problems provided and submit their solutions for Google to test. The submissions that pass Google's tests are considered valid and are published online. The first problem from the fifth round of GCJ from 2011 to 2014 is used as early rounds have many submissions to create a reasonably scoped experiment. The details about the problem and submissions are in Table 3.1. Overall in this study, 247 projects are considered; 170 from Java and 77 from Python. The 170 Java GCJ submissions contain 885 methods and generated 111,203 Java snippets most of

Table 3.1 Projects used in this study with the number of valid submissions in both Java and Python.

Year	Problem	ID	Java	Python
2011	Irregular Cake	Y11R5P1	48	16
2012	Perfect Game	Y12R5P1	47	24
2013	Cheaters	Y13R5P1	29	19
2014	Magical Tour	Y14R5P1	46	18
Total			170	77

which are “invalid” syntactically or contain compilation errors. After removing the invalid snippets and subsequent transformation, SLACC generated 19,188 Java functions.

The 77 Python submission contains 301 methods and generated 82,362 Python functions. Like Java, most of these segments are invalid due to syntactic or compilation errors. After removing the invalid snippets and subsequent transformation generates 17,215 Python functions.

The code, projects and execution scripts for the project can be found in our GitHub Repository [149].

3.5 Experimental Setup

The experiments were run on a 16 node cluster with each node having a 4-core AMD opteron processor and 32GB DDR3 1333 ECC DRAM. Our experiments have four hyper-parameters

- Minimum size of snippet (MIN_STMT - section 3.2.1): This is set to 2 to capture snippets with interesting behavior.
- Maximum number of arguments (ARG_MAX - section 3.2.2): This is set to 5. Hence if a snippet has more than 5 arguments, it is omitted from the experiments.
- Number of executions (section 3.2.4): Each snippet is executed with 256 generated inputs (section 3.2.3); see section 6.1.1 for details on this choice.
- Similarity Threshold (SIM_T - section 3.2.5): This is set to 1.0 which implies that two functions are only considered to be clones if for *all* inputs they generate the same outputs.

Sensitivity to the number of executions and ARG_MAX is explored and discussed in Sections 6.1.1 and 6.1.3 respectively.

3.6 Metrics

The study uses three metrics primarily to address the research questions.

- **Number of Clusters:** A cluster is a collection of functions with a common property (i.e., type I-IV similarity). This metric is the number of clusters generated by a clone detection algorithm. This is represented as $|\text{Clusters}|$, # Clusters or #C.
- **Number of Clones:** A function that belongs to a cluster is called a clone. This metric is the total number of functions in all the clusters generated by a clone detection algorithm. This is represented as $|\text{Clones}|$, # Clones or #M.
- **Number of False Positives:** A false positive is a cluster which contains one or more functions which does not adhere to the similarity measure of the cluster. This is represented as $|\text{False Positive}|$, # False Positives or #FP.

3.7 Baselines

To answer RQ1, RQ2, and RQ3, the following baseline techniques are used to illustrate the capabilities of SLACC.

3.7.1 RQ1: HitoshiIO

As a baseline, the closest technique to SLACC, HitoshiIO [102] is used. This tool identifies functional clones for Java Virtual Machine (JVM) based languages such as Java and Scala. It uses in-vivo clone detection and identifies potential inputs of each code fragment and then executes them with existing workloads like tests or entry points (For example, the `main` function in Java) to collect values from their outputs by inserting instrumentation code in the form of control instructions [158] in the application's bytecode to record input and output values at runtime. Inputs and outputs are observed using the existing workloads, which allows it to observe behavior and identify clones in code for which input generators cannot generate inputs. The methods with similar values of inputs and outputs during executions are identified as functional clones. HitoshiIO considers every method in a project as a potential functional clone of every other method, recording observed inputs that can be method parameters or global state variables read by a method, and observed outputs that are externally observable writes including return values and heap variables. It subsequently uses a relaxed similarity comparison, enabling efficient detection of code that has very similar inputs and outputs, even when the exact data structures of those variables differ and returns pairs of clones. For comparison against SLACC, the pairs are grouped into clusters as follows: two pairs of clones are grouped into a cluster if both the pairs have a common function between them. More simply, consider two pairs of semantic code clones (A, B) and (C, A). A, B and C form a cluster of clones since the two pairs contain the common function C.

Like the similarity threshold `SIM_T` in SLACC, HitoshiIO has a similar parameter that provides a lower bound on how similar two methods must be to be considered a functional clone. As with SLACC, HitoshiIO also has a parameter for an upper bound on the number of IO profiles considered for each method.

An existing and public implementation of HitoshiIO is used for this study.² The workload used to benchmark HitoshiIO with GCJ are the sample test input files. GCJ provides only two sample input files for a validating a submission. However, in SLACC each method was executed 256 times. To create a balanced benchmark, the test input files are randomly fuzzed 32 times before sending it to HitoshiIO. Note that fuzzing the files 256 times led to crashes for large number of inputs in the clone-detection phase of HitoshiIO.

3.7.2 RQ2: Automated AST Comparison

HitoshiIO can detect semantic code clones in Java. However, it cannot be extended to support Python as its modules are hardwired to Java. The literature search failed to find a prior work to detect semantic code clones in dynamic languages. Hence, SLACC was benchmarked for dynamic and cross-language clones by matching the Abstract Syntax Trees (ASTs) as a proxy for similarity. This technique has been adopted by many graph-based (an example of type-III clone) code clone detection techniques in C [11, 18, 38] and Java [35, 38].

Like SLACC, the first phase of the AST comparison segments the code into snippets. Next an AST is generated for the snippets. JavaParser [87] tool and Python AST [152] module used to construct the ASTs in the respective languages. Similarity is measured by matching the ASTs. For clones in the same programming language (RQ1, RQ2), the ASTs are compared and considered to be type-III clones if the ASTs are equivalent or have a difference of at most one node.

3.7.3 RQ3: Manual Cross-language AST Comparison

The AST comparison approach cannot be adopted for cross-language clones (RQ3) due to the difference in format of the ASTs for both the languages. In this case, conservatively, cross-language snippets were sampled with extremely similar outputs and manually verified the ASTs for similarity. To do this, 1 million pairs of a Java function and a Python function were randomly sampled. If the input and output types are compatible, and the outputs are the same for the same inputs or off by a *consistent* value, the ASTs are manually evaluated for similarity. *Consistency* is determined based on the output type. Values of primitive types are consistent if they have a constant difference (for Boolean or Numeric values), constant ratio (for Boolean or Numeric values) or constant Levenshtein distance [128] (for Strings) between the outputs. Objects are consistent if each member of the object is consistent. Finally, two arrays are consistent, if all the corresponding members of the array are consistent.

For example, given two methods, `int A(int x)` and `def B(y)`, if $A(1) = 1, B(1) = 9, A(2) = 2,$ and $B(2) = 18$, then $A()$ and $B()$ are similar since their outputs have a constant ratio (9). Of the 616 similar pairs, all had identical ASTs or had a difference of at most one node, making them type-III clones.

²github.com/Programming-Systems-Lab/ioclones;
Commit hash: aa5b5b3; Dated: 05/06/2018

3.8 Precision Analysis

SLACC and HitoshiIO are both clustered using IO relationships of the functions. However, given a different set of inputs, some functions in a cluster might produce a different set of outputs such that they are not clones; such clusters are marked as *false positives* and considered invalid. False positives are identified at the cluster-level in keeping with prior work [47].

To detect false positives, SLACC clusters are re-executed on a new set of 256 inputs generated using random fuzzing [47] based on a triangular distribution, and clustered. If any method in a cluster is not grouped into the same cluster using the new input set, the whole cluster is marked as a false positive. We see that the number of clusters and false positives is relatively stable above 64 inputs (section 6.1.1).

HitoshiIO needs an input file with test cases to execute the program. To detect false positives in HitoshiIO, the test input files are randomly fuzzed 32 times (section 3.7.1) to generate a new test file that is 32x the size of the original, and then re-execute HitoshiIO. Clone pairs are clustered and false-positives are detected when a new cluster does not match an original cluster, as done for SLACC.

False positives in clusters generated by AST comparisons are identified in a similar manner to SLACC. ASTs in the clusters are first converted to functions (as described in section 3.2.2). The functions are re-executed on 256 inputs like SLACC clusters and checked for false positives. Any cluster that contains a different method after execution is marked as a false positive.

3.9 Results

The results show that SLACC identifies more method level clones compared to prior work and with higher precision (RQ1), successfully identifies clones in dynamic typed languages (RQ2), and successfully detects clones between Java and Python (RQ3).

3.9.1 RQ1: Static Typed Languages

The 885 Java methods generated 19,188 Java functions for analysis. SLACC was able to support 691 of the 885 Java methods. From the 691 whole methods, 18,497 functions are derived into partial method snippets. Of the total generated functions, 4,180 (22%) are clones resulting in 632 clusters. These 4,180 clones derive from 4,038 partial-method snippets and 142 whole methods. They are called *statement level* clones and *method level* clones, respectively.

3.9.1.1 Method level clones

HitoshiIO and SLACC identify clones based on the IO relationship between the functions. The base-line HitoshiIO can detect only similar methods unlike SLACC which can detect similar statements as well. Thus, SLACC is benchmarked against HitoshiIO by comparing clones detected by SLACC at a method level granularity. All 885 Java methods are provided to HitoshiIO, which groups 43 of the

SLACC_{stmt}

```
1 import Y14R5P1.stolis.MMT3 // Parent Class MMT3
2 public static int func_a(BufferedReader br){
3     // Snipped from Y14R5P1.stolis.MMT3.main()
4     if (!MMT3.in.hasMoreTokens())
5         MMT3.in = new StringTokenizer(br.readLine());
6     int a = Integer.parseInt(MMT3.in.nextToken());
7     return a;
8 }
```

SLACC_{method}

```
1 import Y12R5P1.xiaowuc.A // Parent Class A
2 public static int func_b(Scanner in) {
3     // Y12R5P1.xiaowuc.A.next()
4     while (A.tok == null || !A.tok.hasMoreTokens()) {
5         A.tok = new StringTokenizer(in.readLine());
6     }
7     return Integer.parseInt(A.tok.nextToken());
8 }
```

HitoshiIO

```
1 public static int func_c(StreamTokenizer in) {
2     // Y11R5P1.burdakovd.A.nextInt()
3     in.nextToken();
4     return (int) in.nval;
5 }
```

```
1 public static int func_d(StreamTokenizer in) {
2     // Y11R5P1.Sammarize.Main.next()
3     in.nextToken();
4     return Integer.parseInt(in.nval);
5 }
```

```
1 import Y14R5P1.eatMore.A // Parent Class A
2 public static int func_e(Scanner in) {
3     // Y14R5P1.eatMore.A.next()
4     A.in = in;
5     return Integer.parseInt(A.nextToken());
6 }
```

```
1 public static int func_f(Scanner sc) {
2     // Snipped from Y11R5P1.dooglius.A.go()
3     int next = sc.nextInt();
4     return next;
5 }
```

Figure 3.4 Semantic clusters detected by HitoshiIO, SLACC on method level (*SLACC_{method}*) and SLACC on statement level (*SLACC_{stmt}*). The cluster contains functions that take an object that reads a file and returns the next Integer token.

Table 3.2 Number of whole method clones identified by HitoshiIO(H), SLACC(S) and both the approaches, after accounting for false positives.

Problem	HitoshiIO(H)	SLACC(S)	$H \cap S$
Irregular Cake	3	44	3
Perfect Game	4	35	4
Cheaters	4	21	4
Magical Tour	9	35	9
Total	20	135	20

methods into 13 clusters. False positives were identified for 9 of the 13 clusters (precision=30.7%).³ The remaining valid clusters from HitoshiIO contain 20 methods. From the 691 Java methods, SLACC detected 142 methods, grouped into 15 clusters. False positives were identified for 2 of the 15 clusters (precision = 86.7%). The remaining valid clusters for SLACC contain 135 methods.

Table 3.2 shows the numbers of valid clusters for each approach, as well as their intersection. All valid clusters from HitoshiIO are contained within the valid clusters for SLACC, ($H \equiv H \cap S$), demonstrating that among the valid clones, SLACC subsumes HitoshiIO for this experiment. However, the low precision for HitoshiIO may be due to the use of limited inputs or the execution context, so further investigation is needed for generalization of this result.

An example of a cluster that contains methods from both SLACC and HitoshiIO is shown in Fig. 3.4. The cluster contains functions that take an object that reads a file and returns the next Integer token. Functions `func_c` and `func_d` are clones detected by HitoshiIO. Within the same cluster, `SLACCmethod` additionally identifies two more method level clones that were not detected by HitoshiIO: `func_b` and `func_e`.

3.9.1.2 Statement level clones

Additionally, SLACC identifies 624 clusters with 4,038 statement level code clones. Of these, 48 clusters are false positives (precision=92.3%). The large number of code clones is intuitive because each method can contain multiple modular functionalities. That said, it should be noted that the higher precision for statement level clusters would lead us to believe that detecting clones for succinct behavior is more accurate.

Statement level clones can be clustered with whole method clones. For example, in Fig. 3.4, `SLACCstmt` represents a SLACC cluster based on partial methods: `func_a` and `func_f` are functions segmented from the main method in class `Y14R5P1.stolis.MMT3` and the `go` method in `Y11R5P1.dooglius.A`, respectively.

³False positive rates in the original HitoshiIO paper [102] are computed at the pair-level rather than cluster level and used student opinions rather than code behavior, which may account for the relatively low precision reported here.

Table 3.3 # of Java, Python and Cross language clusters detected by SLACC compared against AST (Type-III) clusters.

	Java		Python		Java + Python	
	SLACC	AST	SLACC	AST	SLACC	AST
# Clusters	632	6122	482	3971	34	616
# Valid	584	226	421	181	32	25
Precision	92.4	3.7	87.3	4.6	94.1	4.1

```

1 def func_db8e(a):
2     n = len(a)
3     sum0 = [0] * (n + 1)
4     for i in xrange(n):
5         sum0[i + 1] = sum0[i] + a[i]
6     allv = sum0[-1]
7     return allv

1 def func_43df(items):
2     _sum = sum(items)
3     j = len(items) - 1
4     return _sum

```

Figure 3.5 Semantic cluster of Python functions detected by SLACC. The cluster contains functions that returns the sum of an input array.

RQ1: Method level clones: SLACC identifies more method level clones compared to HitoshiIO at higher precision. *Statement level clones:* Segmentation of code increases the precision of SLACC and yields a higher number of semantic clones.

3.9.2 RQ2: Dynamic Typed Languages

SLACC identified that 3,135 (18.2%) of the 17,215 extracted Python functions had clones which resulted in 482 clone clusters. Of these 482 clusters, 421 are valid, resulting in precision of 87.3%. As a baseline, using the same Python functions, type-III clones were systematically identified. There exists 3,971 clusters, of which 181 are valid (4.6% precision); these results are shown in the *Python* column of Table 3.3, where *AST* shows the type-III clones. For sake of comparison, the experiment was repeated for Java clones; a similar differential between SLACC and AST precision was observed (92.4% vs. 3.7%). This would lead us to believe that traditional methods that detect syntactic type-III clones cannot be used for behavioral similarity, despite successful applications for single languages in prior work for identifying libraries with reusable code [24], detecting malicious code [40], catching plagiarism [14] and identifying opportunities for refactoring [82].

When these clusters are validated, 61 of the 482 SLACC clusters (12.8%) were deemed to be false positive. This is more than the percentage of false positives in Java (7.3%), but by executing the

```

1  static long func_3b0e (Long[] x2) {
2      Long res = null;
3      Long[] arr = x2;
4      int len = arr.length;
5      for (int i = 0; i < len; ++i) {
6          long xx = arr[i];
7          if (xx >= res)
8              continue;
9          res = xx;
10     }
11     return res;
12 }

1  def func_6437 (y):
2      ymin = min (y)
3      count = 0
4      return ymin

```

Figure 3.6 Semantic cluster of a Java function and a Python function detected by SLACC. The cluster contains functions that returns the minimum value in an input integer array.

functions over a larger set generated arguments, the subsequent clustering could yield more robust results. It should be noted that SLACC detected fewer clones in Python compared to Java. This could be due to dynamic typing and limited support offered in case of external modules when it comes to Python.

An example of Python clones identified by SLACC can be seen in Fig. 3.5. Both the functions in this example compute the sum of an array. `func_db8e` uses a loop that maintains the running sum where each index in the array contains the array sum until that index. The last index of the array would contain the array sum and is eventually returned. In contrast, `func_43df` uses the `sum` library function to perform the same task.

RQ2: SLACC can successfully identify code clones for dynamic typed languages with high precision (87.3%).

3.9.3 RQ3: Across Programming Languages

In this section, SLACC is executed on the Java and Python projects from GCJ. From 36,403 extracted snippets, SLACC identified 131 Java and 48 Python functions clustered into 34 cross-language clusters (single-language clusters are omitted from the RQ3 analysis). On validation, we see that 2 of these 34 (5.8%) clusters are false positives which is better than the percentage of false positives found in Java and Python independently. That said, SLACC would produce more clusters when support for the languages is broadened.

It should be noted that since there were restrictions on generated arguments made to address this research question and the functions should be considered code clones if these restrictions

are ignore. For example, `long` in Java and `int` in Python have the same range between -2^{63} and $2^{63} - 1$, the methods in Fig. 3.6 were judged to be clones. But, if the input argument of function `func_3b0e` had an `int` ($-2^{31}, 2^{31} - 1$) datatype, then the two functions would not be clones as the Python function would operate over a larger input range.

616 type-III clusters were discovered by comparing the ASTs of Java and Python snippets (Table 3.3), of which 25 clusters are valid (4.1% precision). It should be noted that this is a conservative precision estimate; the baseline was created by starting with close behavioral matches, hence giving the AST analysis a slight edge on precision (section 3.7.2).

An example of a pair of Java-Python clones can be seen in Fig. 3.6. `func_3b0e` is a Java function that uses a loop to find the minimum in an array while `func_6437` is a Python function uses the inbuilt `min` function in Python.

RQ3: SLACC succeeds in identifying clones between programming languages irrespective of their typing.

3.10 Limitations and Threats

Threats to external validity include the focus on two languages as instances of static and dynamic typing, so results may not generalize beyond Java and Python. The use of GCJ code may not generalize to more complex code bases. Threats to internal validity include that for RQ3, where the AST matching is aided by starting with behavioral clusters and then determining if the ASTs are similar; in this sense, the precision of cross-language AST matching is overestimated.

Our implementation of COSAL has the following limitations:

Dynamic Typing. COSAL does not support two primitive types `long` and `complex` for Python. That being said, the GCJ projects used in this study were and do not explicitly use these values in the source code and they are not present in the input file used by the baseline HitoshiIO. Further, in case of a failure to identify the type of a function argument, the function was fuzzed with arguments of all supported types. In this study, only primitive types and the simple data-structures `tuple`, `set`, `list` and `dict` is supported. Support for other sophisticated data-structures can be incorporated by extending the existing COSAL API with instructions in the wiki [149].

Hyper-parameter bias. COSAL uses four hyper-parameters whose values (see section 3.5) were chosen based on research evaluation and engineering limitations.

- `MIN_STMTS`: Although, this was set to 2, COSAL analyzes singular statements if it contained nested statements (For eg. block, loop and conditional statements). This parameter simply vets out single line declaration and assignment statements as the detected clusters would contain trivial clones.
- `ARGS_MAX`: The choice of this value influences the space of generated functions by an order of $O(ARGS_MAX!)$ and could lead to disk space and memory explosion. Hence, based on

experimental validation, *ARGS_MAX* was set to 5.

- **Number of Executions:** Execution of snippets is the biggest bottleneck in COSAL's workflow. We set this to 256 for each snippet based on the available hardware capacity. Larger values would result in better quality clusters with lower false positives. That said, COSAL uses an order of more executions in its evaluation compared to other clone-detection methods [47, 56].
- **SIM_T:** This is set to 1.0 to ensure maximum stability of the clusters and provide a tight benchmark against the baseline HitoshiIO. Lower values of *SimT* would relax the measure of similarity and should be adjusted based on the applications of COSAL (section 6.1.6).

Unsupported Features. Although COSAL supports Object Oriented features such as inheritance and encapsulation, it is limited to objects derived from primitive types. Hence, the current version of COSAL cannot scale to more sophisticated objects like Threads and Database Connections. Similarly, for Python SLACC does not support modules like generators and decorators. Nevertheless, it would be possible to support these features with more engineering effort.

Dead Code Elimination: In the code-clone examples of Fig. 3.5 and Fig. 3.6, we see the presence of lines of code that do not influence the return value i.e., Dead Code. At the moment, the functions do not fail due to dead code but eliminating them would make the functions more succinct and comprehensible. This will be an avenue for future work for specific applications of COSAL.

CHAPTER

4

CODE-TO-CODE SEARCH ACROSS LANGUAGES

This work is published at ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE) 2021 in collaboration with Dr. Kathryn T. Stolee [139].

4.1 Motivation

Code to code search has multiple applications in software engineering. For example, while learning a new programming language, students prefer looking up similar or alternate implementations of a functionality [95]. Similarly, code to code search is used in identifying candidate code snippets which can be used in Automated Program Repair [48]. The state of art techniques on code search and clone detection use individual similarity metrics to compare code snippets. Static similarity metrics are typically based on intermediate representations like Tokens [14, 26, 30], AST [18, 38] or Program Dependency Graphs [43, 61]. Such techniques have two primary limitations. First, due to the reliance on syntactic features, the approaches cannot be scaled for code to code search across programming languages [102]. Second, due to the dependency of similarity measure on the static properties of code, they miss finding snippets which have similar behavior even if their code is syntactically dissimilar [56]. Although fewer in number compared to static similarity based approaches, dynamic similarity based approaches have greater success in identifying snippets that are syntactically different but have similar behavior. These approaches rely on Input-Output behaviour [47, 102, 134] or intermediate execution states [54, 71, 101]. As a result, these approaches

```

1 List<Integer> getEvens(int max) {
2     List<Integer> evens = new ArrayList<>();
3     for(int i = 0; i < max; i++)
4         if (i % 2 == 0)
5             evens.add(i);
6     return evens;
7 }

```

(a) Java: **for** loop to populate an array of even numbers

```

1 def all_odds(n):
2     odds = []
3     for i in xrange(n):
4         if i % 2 == 1:
5             odds.append(i)
6     return odds

```

(b) Python: **for** loop to populate an array of odd numbers

```

1 List removeOdds(List<Integer> nums) {
2     return nums.stream().filter
3         (i -> i % 2 == 0).collect();
4 }

```

(c) Java: **filters out odd numbers from an input List**

```

1 Integer[] func(int x) {
2     int[] n = IntStream.range(0, x)
3         .toArray();
4     List<Integer> e = new ArrayList<>();
5     for (int i=0; i<n.length(); i++)
6         if (n.get(i) % 2 == 0)
7             e.add(n.get(i));
8     return e.toArray();
9 }

```

(d) Java: **List of even numbers using external libraries**

```

1 from functools import ifilter
2 def get_evens(max):
3     # Filter out odds
4     return ifilter(lambda x: x%2, range(max))

```

(e) Python: **list of even numbers using external library**

```

1 def even_nums(max_val):
2     nums = range(max_val + 1)
3     return [i for i in nums if i % 2 == 0]

```

(f) Python: **list of even numbers using list-comprehension**

Figure 4.1 Different functions to return a filtered array of numbers implemented in Java and Python. The code in (a), (d), and (e) are functionally identical.

are not scalable for practical use due to the need for executable code, exhaustive test cases and dependency libraries [56].

Consider the code snippets in Fig. 4.1 which represents six functions that return a filtered array of numbers. Fig. 4.1a is Java code that takes an integer input *max*, and returns an array of even numbers between $[0, max)$. Fig. 4.1b is a Python function that takes an integer argument and returns a list of odd numbers between $[0, max)$. Fig. 4.1c is a Java function that takes a list of integers and removes all the odd numbers from it. Fig. 4.1d creates an array of integers ranging from $[0, x)$ but using the `IntStream` library introduced in Java 8. The function then returns all the even numbers from the array using a `for` loop. Functions Fig. 4.1e and Fig. 4.1f are Python functions that return an array of even numbers from $[0, max)$ and $[0, max_val]$ respectively. Fig. 4.1e uses an external library `functools.ifilter` and Fig. 4.1f uses a `list-comprehension`.

Three of the functions are behaviorally identical which takes an input integer and returns an array of even integers: Fig. 4.1a is a Java function which uses a `for` loop; Fig. 4.1d uses the stream library from `Javav8`; Fig. 4.1f is a Python function which uses a filtered `list-comprehension`. Fig. 4.1b is a Python function that takes an integer *max* and returns a list of odd numbers between $[0, max)$.

To statically identify behaviorally identical Python snippet for Fig. 4.1a, a code-to-code search engine should support both the programming languages Java and Python. Using a purely token based

approach for cross language search will not be very helpful in this case primarily due to two reasons. First, the syntactic features of each language will skew the cross language search. For example, since Java is static typed, variables are declared with the datatype, while variables in Python lack these tokens due to dynamic typing in Python. Second, even if language specific keywords are ignored, there is an over reliance on the names of variables and libraries to infer behavioral context which may not always succeed. For example, Fig. 4.1a uses `evens` to denote a list, while Fig. 4.1f uses `nums` to represent the same array.

On the other hand using an AST based approaches to identify similar code structures across languages is also not consistently viable due to the language specific constructs. For example, Fig. 4.1a uses a traditional for loop to populate the list while Fig. 4.1f uses `list-comprehension` which is a pythonic construct to perform the same task. The nearest match with respect to similar construct would be Fig. 4.1b since it also uses a for loop. That said, the functions Fig. 4.1a and Fig. 4.1b are behaviorally different, since the former uses a `for` loop to populate an array of even numbers while the latter populates an array of odd numbers. In such cases, a dynamic approach based on IO similarity should identify Fig. 4.1a and Fig. 4.1b as behavioral snippets.

Behavioral approach also has its limitations. For example, `IntStream` from Fig. 4.1d is specific to Java v8 and above. Similarly `xrange` from Fig. 4.1f is specific to Python v2.x. Hence the right version of the language and libraries is a prerequisite and in many cases a major bottle neck for dynamic similarity.

Hence like the *no free lunch* theorem, there is no single best technique for code search. It depends on multiple varying criteria which cannot be generalized for all cases. Hence, we need a code-to-code search tool that enables search based on multiple search similarities. That said, combining different similarities has its own limitations like optimizing for weights for each similarity or heavy influence of one specific similarity. This work overcomes these limitations by finding similar code based on multiple different (or available) similarity measures and ranking them using non-dominated sorting. Using such an approach enables cross-language and within-language search which is not bound to one specific search similarity.

4.2 COSAL

This work presents Code-to-Code Search Across Languages, or COSAL for code-to-code search within or cross-languages. As prior approaches to code-to-code rely on machine learning [124, 125] and are thus potentially prone to overfitting. This risk is removed by using SLACC as a dynamic similarity metric to compare behavioral similarity between code snippets. However, as SLACC only clusters executable code, its application in search is limited to executable code. Prior approaches to code-to-code search allow for unexecutable code as queries. To make the search more practical, COSAL uses non-dominated sorting that considers dynamic behavior using SLACC (section 4.2.3) and static code similarities based on code contexts (section 4.2.1), and ASTs (section 4.2.2), thus providing search results even with dynamic analysis falls short.

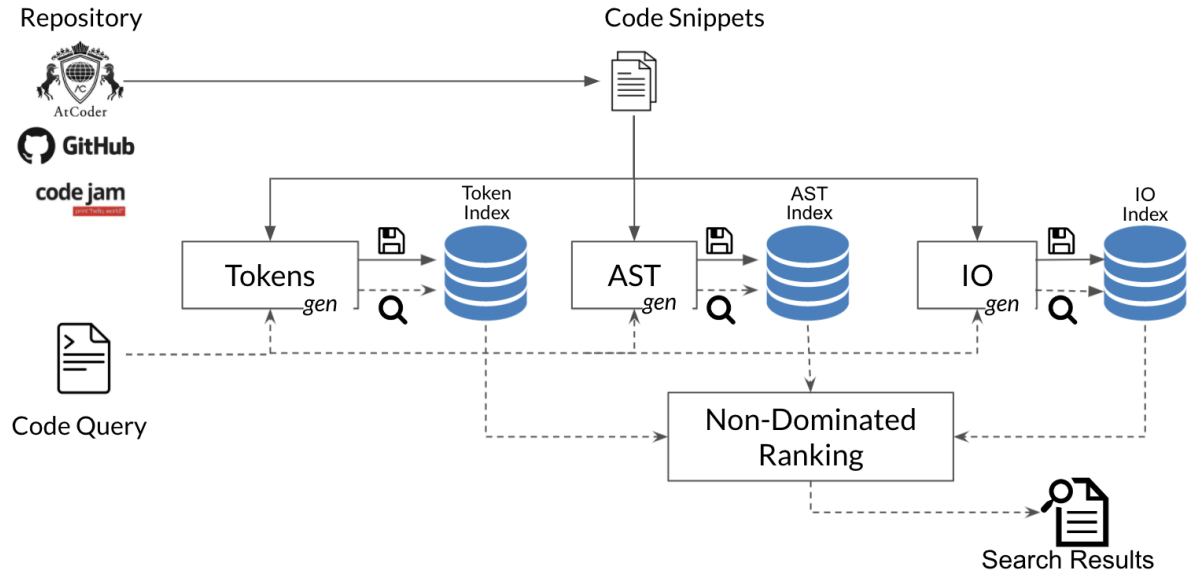


Figure 4.2 Overview of COSAL

Fig. 4.2 depicts the general workflow of code-to-code search using COSAL, which includes a one-time indexing of the *Source Code Repository* and the process of retrieving *Search Results* based on a *Code Query*.

1. Offline, a repository is crawled to extract snippets of code (e.g., GitHub, submissions from programming contests like AtCoder or Google CodeJam, or a local File System.)
2. The extracted snippets are used to create an *Index* for each of the following:
 - (a) *Tokens* representing names and libraries used in the code (section 4.2.1).
 - (b) A language-agnostic *AST* represents the code structure (section 4.2.2).
 - (c) If the code can be executed, the *IO* behavior is recorded based on grey-box fuzzing [134] (section 4.2.3).
3. During search, a *code query* is processed in the same manner as each code snippet, according to Steps (a)-(c), to gather *Tokens*, *AST*, and *IO* information.
4. *Multi-Objective Search* identifies *Search Results*, which are ranked and returned to the user (section 4.2.4)

To illustrate COSAL, we will reuse examples of code from Fig. 4.1.

4.2.1 Token-based Search

Fragments of code that are contextually similar often use similar variable names or libraries [31], though the naming conventions vary by language. For example, Java primarily uses `camelCase`

Table 4.1 Tokens for functions from Fig. 4.1

Fig.	Lang.	Tokens
4.1a	Java	getevens, get, max, arraylist, list, add, integer, array, evens
4.1b	Python	allodds, xrange, odds, append
4.1c	Java	filter, collect, removeodds, stream, nums, integer, remove, odds, list
4.1d	Java	func, intstream, stream, range, toarray, array, list, integer, arraylist, length, add
4.1e	Python	ifilter, filter, range, getevens, get, max, functools, evens, odds
4.1f	Python	even, evennums, nums, max, val, maxval, range

conventions while Python uses `snake_case`. Developers tend to describe the code in comments based on the functionality. Context from source code is inferred by extracting non-language specific tokens from source code and comments using the following approach:

1. Extract a complete list of tokens from code and comments.
2. Remove language-specific keywords obtained from the language documentation [142, 151]. For example, java tokens `function` and `static`, and Python tokens `def` and `assert`, are all removed. Remove frequently-used words used in a programming language. These are not keywords but are part of common coding conventions, for example, in Python, the token `self` is used commonly to denote the class object.¹
3. Remove common stopwords from the English Vocabulary [45], such as `does` and `from`.
4. Split tokens to address language-specific nomenclature. Variables are typically named using `camelCase` in Java and `snake_case` in Python. Such tokens are split into {"camel" and "case"} and {"snake" and "case"}, respectively.
5. Remove tokens of length less than `MIN_TOK_LEN`.
6. Convert all the tokens to lower case.

A repository of code is tokenized using the above approach and stored in an ElasticSearch [88] index. For a user's code query, the tokens generated from the indexing approach are looked up in the search index and the best matched results are returned using the *token similarity distance* (d_{token}). This distance is the same as the Jaccard Coefficient [75] and is defined as follows:

$$d_{token} = \frac{|tokens_{query} \cap tokens_{result}|}{|tokens_{query} \cup tokens_{result}|}$$

¹Complete lists of the removed tokens are available [135].

Table 4.2 Similarity metrics for Java(*J*) and Python(*P*) functions from Fig. 4.1. High similarity is associated with high values (\uparrow) of d_{token} , low values (\downarrow) of d_{AST} , and high values (\uparrow) of d_{IO} .

Snip 1	Snip 2	$\uparrow d_{token}$	$\downarrow d_{AST}$	$\uparrow d_{IO}$	
				($\mathbf{s}_1, \mathbf{s}_2$)	($\mathbf{s}_2, \mathbf{s}_1$)
J-4.1a	P-4.1b	0.0	1	0.5	0.5
J-4.1a	J-4.1c	0.125	23	0.0	0.0
J-4.1a	J-4.1d	0.333	16	1.0	1.0
J-4.1a	P-4.1e	0.286	19	1.0	1.0
J-4.1a	P-4.1f	0.067	7	0.5	0.5
P-4.1b	J-4.1c	0.083	23	0.0	0.0
P-4.1b	J-4.1d	0.0	17	0.5	0.4
P-4.1b	P-4.1e	0.083	19	0.0	0.0
P-4.1b	P-4.1f	0.0	8	0.0	0.0
J-4.1c	J-4.1d	0.176	33	1.0	0.5
J-4.1c	P-4.1e	0.125	8	0.0	0.0
J-4.1c	P-4.1f	0.067	21	1.0	0.5
J-4.1d	P-4.1e	0.053	29	1.0	1.0
J-4.1d	P-4.1f	0.059	21	0.5	0.5
P-4.1e	P-4.1f	0.143	17	0.0	0.0

d_{token} will range from [0.0, 1.0]. Larger values of d_{token} indicate higher token similarity between the query and result candidate.

For the functions in Fig. 4.1, the generated tokens using this approach are shown in Table 4.1 and the token similarity distance for each pair of functions is shown in Table 4.2 (d_{token}). Note that none of the functions in Fig. 4.1 have code comments; in our implementation the comments are included in the token list. Based on this metric, if the Java function Fig. 4.1a is the query, the best Python result would be Fig. 4.1e ($d_{token} = 0.286$). These functions are behaviorally similar to each other and the tokens extracted from these functions help in identifying this similarity. However, in many cases the token-based scoring cannot yield best results. For example, if the Python function Fig. 4.1e is the query, the best Java function would be Fig. 4.1c, which differs in functionality to Fig. 4.1e.

Token-based analysis relies on self-describing snippets; the choice of variable names, function names, libraries used, and comments all impact the the results. Not all code snippets adhere to intuitive naming conventions. For example, in Fig. 4.1d, the programmer chose very generic names. Despite this limitation, in section 4.9.1 we see that the tokenization approach adopted by COSAL yields more precise results compared to full text search.

4.2.2 AST-based Search

COSAL uses a tree based representation to syntactically compare snippets of code. This is a challenging across languages since there is no generic AST representation of code that encompasses

syntactic features of different languages. Traditional AST parsers like Antlr, JavaParser, python-ast modules use different grammars to denote similar features as well. For example, a function node in JavaParser is represented as `MethodDeclaration` while the python-ast parser represents the node as `FunctionDef`. As a result, to compare ASTs of different programming languages, we require a mapping scheme between each pair of programming languages.

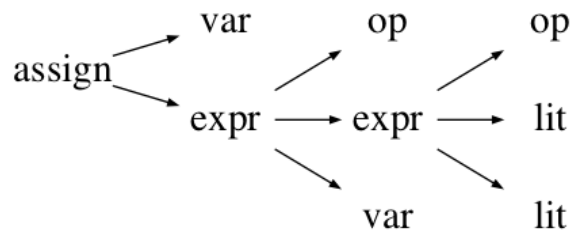


Figure 4.3 Normalized AST for the Java statement `'int x = y + 15 * 2.0;'`

To account for better scalability to additional programming languages, COSAL uses a parser for a generic AST. By mapping the ASTs for Java and for Python onto the generic AST, we can compare across these languages (see section 6.2.3) for a discussion on scalability). The generic AST contains a superset of the language features, as follows:

- **Common control structures:** Control structures are simplified to make them applicable cross-language. For example, the `loop` node is used for the following example Java constructs: `for`, `forEach`, `while`, `do-while` and `for-each`; and example Python constructs: `for`, Python `list-comprehension` and Python `dict-comprehension`.
- **Normalizing Variable:** All variables are denoted as `var` nodes. For example, the statement `int x = y + 15 * 2.0;` is converted to the AST shown in Fig. 4.3 where all the variables are denoted as `var` node.
- **Normalizing Literals:** All literals are denoted as `lit` nodes. This can also be seen in Fig. 4.3 where the literal `15` and `2.0` are represented as as `lit` nodes.
- **Normalizing operators:** All operators are also denoted as `op` nodes.
- **Language specific features:** If a feature is implemented in only one language, a custom node is created for that feature. For example, `switch` statement are specific to Java and not supported in Python. As a result, a custom node `switch` is created for this statement.

The representation of the generic AST is based on our intuition, and there may be more effective or efficient representations; future work will explore various representations of the AST and the impact on the objective effectiveness.

Once the ASTs are constructed, the similarity between them is computed using the tree-edit distance [8] (d_{AST}). The algorithm computes the minimum number of edits required to transform an ordered labeled tree to another ordered labeled tree in sequential time. d_{AST} will range from $[0, \infty)$. Lower values of d_{AST} are associated with higher similarity.

For the functions in Fig. 4.1, the generated ASTs are shown in Table 4.3 and the AST edit distance for each pair of functions is shown in Table 4.2 (d_{AST}). Based on d_{AST} , if the Python function Fig. 4.1f is a query, the best Java result would be Fig. 4.1a ($d_{AST} = 7$). Notice that when Fig. 4.1f was the query using token-based search, all three Java functions had very low values of d_{token} (and thus low token-based similarity). However, using d_{AST} , the Python search query and the Java search result (Fig. 4.1a) have a substantially smaller edit distance compared to the other two Java functions. Notably, the syntactic constructs of the two functions are also different. The Python search query (Fig. 4.1f) uses a list comprehension which is a Python feature and the Java search result (Fig. 4.1a) uses a for loop. Identifying the matching search result is possible, since the list-comprehension feature is mapped as a for loop in the grammar for the generic AST of COSAL.

There are cases where a generic AST-based approach is non-ideal. For example, if the Java function Fig. 4.1a is a query, the best Python result would be Fig. 4.1b using d_{AST} alone. This is because both functions use traditional for loops and updates the return array sequentially, and yet, the search result is functionally different from the query. Such scenarios can be handled using dynamic similarity.

4.2.3 Input-Output based Search

Dynamic search in COSAL, is performed by clustering code based on their IO relationship. Consider a query q and a potential search result s . We should not require a perfect matching between the methods in q and s . As a practical example, say a developer is looking for a Python API for *QuickSelect*², which finds the k^{th} smallest number from an array of integers. It has a method that identifies a random pivot in the array and a method that swaps values. However, these methods do not call each other. Thus, to characterize the behavior of this file, we characterize and aggregate the behavior of fragments of the file. Then, when comparing to a custom Python *QuickSort*³ API that has a function to recursively find a random pivot and perform a swap operation, a match is identified even though the number of methods and how they accomplish the same task are different.

To determine the IO relationship between two pieces of code, COSAL uses SLACC Chapter 3. SLACC supports identifying similar code snippets based on segments of functions. SLACC segments code into executable snippets of size greater than `MIN_STMTS` and executed on `ARGS_MAX` arguments generated using a grey box based strategy. The executed functions are then clustered using a representative based clustering approach with a custom similarity measure (*sim*) based on the inputs and outputs of the functions.

SLACC succeeds at finding clones at the method-level and smaller. Notably, it splits methods

²from `org.apache.datasketches`

³`stackabuse.com/quicksort-in-python`

into smaller segments prior to execution. COSAL does not cluster the segmented functions. Instead, it stores the inputs and outputs for each snippet in the *IO Index* (Fig. 4.2). The similarity information between a query q and a potential search result s is aggregated by finding segments in s with the highest similarity for each segment from q , and then taking an average. In this way, COSAL can identify similarity at a file-level to mimic industrial use cases [92].

Let Q and S be sets of segments identified by SLACC from q and s respectively. The IO similarity between q and s is defined as

$$d_{IO}(q, s) = \frac{1}{|Q|} \sum_{q_i \in Q} \underset{s_k \in S}{\text{maximize}} \text{sim}(q_i, s_k)$$

Values of d_{IO} range from [0.0, 1.0]. Higher similarity corresponds to higher values of d_{IO} . Consider an example: let $Q = \{q_1, q_2, q_3, q_4, q_5\}$ be set of five segments and $S = \{s_1, s_2, s_3\}$ be set of three segments identified by SLACC. Segments q_1 , q_2 and q_3 find segments s_1 , s_2 and s_1 to be the most similar, respectively, with similarity scores (sim) of $\text{sim}(q_1, s_1) = 0.8$, $\text{sim}(q_2, s_2) = 0.95$ and $\text{sim}(q_3, s_1) = 0.7$. Segments q_4 and q_5 did not find segments in S with similarity greater than 0.0. In this case, $d_{IO}(q, s) = \frac{0.8+0.95+0.7+0.0+0.0}{5} = 0.49$.

The dynamic similarity between all the functions in Fig. 4.1 is shown in Table 4.2. We noticed in section 4.2.2 that Fig. 4.1a and Fig. 4.1b were very similar to each other based on d_{AST} but functionally different. If the same functions are compared based on behavioral analysis, we see that they are indeed functionally different as $d_{IO} = 0.0$. In contrast, Fig. 4.1d and Fig. 4.1e are similar to each other based on their behavior ($d_{IO} = 1.0$), but relying purely on a static similarity measure would indicate the exact opposite as we can see the pair has a very low value for d_{token} and a large value for d_{AST} .

4.2.4 Non-Dominated Sorting

COSAL uses the Non-Dominated Sorting algorithm from NSGA-II [25], a popular multi objective search technique to rank search results based on d_{token} , d_{AST} , and d_{IO} . The non-dominated sorting algorithm orders results with multiple objectives without aggregating them into a single objective. COSAL uses the algorithm in a novel context; this is the first work that uses multi-objective search for code-to-code search.

Classically, search methods [5, 15] and evolutionary algorithms [13] convert multi-objectives to a single-objective problem [62]. Using such methods is not very optimal as ranking the results would be very subjective if the objectives are independent (which ours mostly are).

COSAL incorporates search as follows:

1. **Individual Search:** For a query, top TOP_K search results are fetched using each similarity measure (d_{token} , d_{AST} and d_{IO}) independently.
2. **Merge:** The individual search results are merged.

3. **Sort:** The merged results are sorted by NSGA-II [25] by measuring the *dominance* of one result over the other.

A search result s is said to dominate a search result t if s is no worse than t in any objective and is better than t in at least one objective. Otherwise, there is a tie. In case of a tie, COSAL selects the point which has the dominant objective closest to the optimal value. For example, consider two search results s and t with $(d_{token}, d_{AST}, d_{IO})$ of $(0.7, 7, 0.9)$ and $(0.8, 7, 0.8)$ respectively; s dominates t on d_{IO} , t dominates s on d_{token} , and s and t have the same value for d_{AST} (note: since t dominates s on one objective and s dominates t on another objective, regardless of the relationship for the third objective, s and t are in a tie). The normalized distance between d_{IO} of s to its optimum value is computed as $\frac{\max(d_{IO}) - d_{IO}(s)}{\max(d_{IO}) - \min(d_{IO})} = 0.1$. Similarly, the normalized distance between d_{tokens} of t to its optimum value is 0.2. Since s is closer to the optimum of its dominant objective (d_{IO}) compared to t to its dominant objective (d_{token}). Hence, s is the better search result.

The application of non-dominated sort to code-to-code search allows COSAL to consider static and dynamic information about a code query and potential search results. It reaps the benefits of dynamic analysis in finding code that behaves similarly, when such code is available, and the benefits of static information when dynamic information is infeasible. It provides results that balance *how code looks* with *how it behaves*, in the spirit of returning code that looks more natural to the user.

4.3 Supporting functional programming

Programming languages are grouped into “paradigms” that differentiate one group of similar languages from the other on a high-level. Some of the major conventional paradigms are imperative, object-oriented (OO), logic and functional.

Python offers a profusion of programming styles like procedural, functional and object oriented and Java is an OO language with Java 8 offering some functional features like lambda functions. Learning programming languages across different programming paradigms is a major challenge in software engineering due to the handling of features like *mutation* and *state* [147]. Traditional similarity metrics based on syntactic tokens features can also not be applied due to vastly different syntactic representation between languages of different programming paradigms. Dynamic similarity based approaches can be used to compare these languages as code snippets will be compared based on behaviour rather than syntactic features. However, functional programming languages have associated challenges like type inference and value based programming.

COSAL can be augmented to support Haskell, a functional programming language. COSAL has three similarity measures based on contexts (d_{token}), structure (d_{AST}) and behavior (d_{IO}). Contextual search is based on extracting language agnostic tokens from code. COSAL adds this support to Haskell using an ANTLR [16] grammar and custom tokenizer. Structural search is based on creating a generic AST and comparing them based on Zhang-Shasha edit distance [8]. Next, the Haskell AST generated using ANTLR is also used to generate the generic AST using a custom language specific adapter. COSAL performs behavioral search using SLACC by comparing the input

output relationships of snippets of code. COSAL extends SLACC to support Haskell by creating a segmentation module to segment out snippets of executable Haskell code and a standalone execution engine [134].

4.4 Research Questions

COSAL aggregates multiple individual cross-language search objectives and sort the results. This leads to the first research question, if a multi-objective approach is necessary.

Research Question 4

Does multi-objective search using tokens, AST and IO yield better results for cross-language code-to-code search as compared to any subset of those objectives?

After validating the choice of a multi-objective search using standard search quality metrics (section 4.7.1), COSAL is compared to the state-of-the-practice search in GitHub Search and ElasticSearch which are based on full text search.

Research Question 5

How effective is COSAL compared to state-of-the-practice public code search tools?

The state-of-the-art in code-to-code search is within-language, but COSAL is a multi-language tool. Still, to compare against the state-of-the-art, this study limits COSAL to within-language and evaluate it against FaCoY [115].

Research Question 6

How effective is COSAL in within-language code-to-code search as compared to the state-of-the-art?

Code-to-code search is often entangled with code clone detection tools. In fact, code-to-code search is commonly used in clone detection due to their similarity [48, 127]. Using COSAL for clone detection, it is compared against ASTLearner [125], CLCDSA [124], and SLACC [134]:

Research Question 7

Can COSAL effectively detect cross-language code clones?

Current cross-language code similarity approaches [124, 125, 138] support languages that have the same programming paradigms. Using COSAL we see how it compares against state-of-the-practice code-to-code search tools GitHub Search and ElasticSearch on languages with different

Table 4.4 Summaries of AtCoder and BigCloneBench datasets

Metric	AtCoder (AtC)			BigCloneBench (BCB)
	Java	Python	Haskell	Java
#Problems / #Features	364	333	42	43
Avg. Solutions per Problem / Feature	57	67	34	1291
#Files	20,828	22,318	1419	55,499
#Methods	81,896	10,020	7024	765,331
Avg. lines/File	51	14	20	278

programming paradigms: Haskell (Functional), Java (Object-Oriented) and Python (Functional and Object-Oriented).

Research Question 8

Can cross-language code similarity approaches be used to compare functional languages to languages with different programming paradigms?

4.5 Data

4.5.1 AtCoder

This study validates search results and compares clone detection tools in cross-language and within-language environments. We require a labeled set of similar code snippets in multiple programming languages for queries and search results. Competitive programming contests satisfy this requirement. Contests such as Google Code Jam [145] and AtCoder [141] have open problems where users can submit their solutions in most common programming languages.

COSAL is validated on the AtCoder [141] dataset since it has been used in evaluating prior code search and clone detection tools [124, 125]. AtCoder is an online platform that holds weekly programming contests with *problems* at different levels of expertise. The solution to a single problem is implemented by multiple participants in many different languages. AtCoder removes solutions which are syntactically incorrect or do not pass the extensive test suite. All the accepted solutions for a single problem implement the same functionality and are behavioral code clones. If a search query and a result belong to the same problem, the result is considered to be *valid* and the query-result pair as valid code clones; the problem solutions are the ground truth in our experiments.

For RQ4-RQ7, the study is limited to the most recent 398 problems which had solutions in Java or Python. RQ8 also considers code snippets from Haskell, a functional programming language to compare code across programming paradigms. For these problems, 44,565 files are from all the *accepted* Java, Python and Haskell solutions. Table 4.4 lists a detailed set of metrics for the study; 307 of the 398 problems have both a Java and Python solutions. The data used in this study is available online [135].

4.5.2 BigCloneBench

BigCloneBench [94] is one of the largest publicly available code clone benchmarks for Java. It contains over 55,000 source code files harnessed from approximately 25,000 open source repositories. This dataset is used as a benchmark for code search [115] and clone detection [99] techniques. The code snippets are categorized into 43 distinct functionality groups like “Connecting to an FTP server”, “Taking a sound source and playing it”, and “Computing prime factors for an Integer” to name a few. Like the AtCoder dataset, query and result snippets belonging to the same functionality are considered as a valid search result.

This study considers fragments of code with at least 6 lines and 50 tokens which is considered a standard minimum clone size for bench-marking [37, 94, 115].

4.6 Baselines

Evaluating RQ4-RQ7 is challenging as the setup is different for each. RQ5 requires baselines from commercial tools while RQ6 and RQ7 require baselines from research tools. For a fair comparison, COSAL is compared to each of the other tools by searching over the same data set. For RQ6 and RQ7 baselines, the source code from the GitHub repositories is used to build the tools for experimentation.

4.6.1 RQ5, RQ8: Text Search

Google search engine is commonly used by developers code search [85]. Developers generally search for code by formulating a query based on keywords, expected code, or exceptions raised. To handle all these cases, Google tokenizes the query and uses page rank to search across the web using full text search. In our study, since the number of files were very large, Google failed to index our code repository after a six week wait. As a result, a custom full text search using ElasticSearch [88] is used which takes in a code snippet, tokenizes the code and identifies results based on Lucene’s Practical Scoring Engine [106]. For this study, each Java and Python file is added to a single ElasticSearch index and searched using the ElasticSearch programmatic search API.

4.6.2 RQ5, RQ8: GitHub Search

GitHub search engine is an IR-based search model over code repositories, including issues, pull request, documentation, and code data. Google search engine typically does not completely index GitHub repositories since a repository typically contains a large number of continuously changing files. Using the built-in code search on GitHub, code can be searched globally across all of GitHub, or searched within a particular repository or organization [156]. Further, GitHub also employs some custom search notations to augment the search experience. However, due to the complexity of code search, GitHub adds restrictions on file size ($\leq 384K B$) and number of files in a repository ($\leq 500,000$). The Java and Python files from the dataset is added to a single GitHub repository and

search within the repository using the GitHub Search API [155]. For all our experiments, the number of files in the repository and the size of each file are well within the limitations of GitHub search.

4.6.3 RQ6: FaCoY

FaCoY [115] is a code-to-code search tool that uses a query alternation approach using relevant keywords from StackOverflow Q&A posts. FaCoY can be modified to change its search database from Q&A posts to custom datasets. To benchmark against COSAL, the search was redirected to the repository of code from AtCoder. In this study, FaCoY does not use StackOverflow as a baseline.

4.6.4 RQ7: ASTLearner

Perez and Chiba developed a semi-supervised cross-language syntactic clone detection method (which will be referred as ASTLearner) [125]. Source code is converted to its AST and a vocabulary is generated for each programming language. Next they train a skip-gram model [72] to vectorize the AST. The vectorized tokens are encoded using an LSTM (Long short-term memory [69]) based encoder. A feed forward neural network classifier using negative sampling is then trained to identify clones. ASTLearner considered pairs of code as clones if the classifier score is greater than 0.5.

4.6.5 RQ7: CLCDSA

Cross Language Code Clone Detection [124], or CLCDSA, uses syntactical features and API documentation to detect cross-language clones. Nine features are extracted from the AST; API call similarity is learned using the API documentation and a Word2Vec [73] model. The vectorized features train a reconfigured Siamese architecture [12] using a large amount of labeled data. CLCDSA uses cosine similarity to detect clones and the best F1 scores were found when the similarity threshold for was 0.5.

4.7 Metrics

The following evaluation metrics are used to evaluate COSAL.

4.7.1 Code Search Metrics

For code search applications (RQ4, RQ5, RQ6, RQ8), *Precision@k*, *SuccessRate@k*, and *MRR* are used. Q is the set of all queries.

Precision@k or $P@k$ is the average percentage of relevant results in the top- k search results for a given query [114, 115]. Mathematically

$$Precision@k = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{|relavent_{i \in k}|}{k}$$

The range is [0.0, 1.0]. Higher values mean more relevant results.

SuccessRate@k or *SR@k* is the percentage of queries where one or more relevant result exists in the top-k search results [114, 123]⁴. For a single query, intuitively, it is measuring whether or not a relevant result was found in the top-k results.

Mathematically,

$$SuccessRate@k = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \chi(BestRank(Q[i]) \leq k)$$

The range is [0.0, 1.0]. The function χ is an indicator function which returns 1 if the input is true and 0 if false. For $k = 1$, *SuccessRate@k* and *Precision@k* are the same. For higher values of k , *SuccessRate@k* indicates whether there is something relevant in the results, and *Precision@k* measures how relevant the k results are on average.

MRR is the Mean of the Reciprocal Rank of the most relevant search result for each query [114, 115, 123]. Mathematically,

$$MRR = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{BestRank(Q[i])}$$

The range is [0.0, 1.0]. Higher values mean relevant results are ranked higher in the results. MRR is limited to the top-300 search results

4.7.2 Clone Detection Metrics

In the case of clone detection [124, 125] (RQ7), *Precision*, *Recall* and *F1* score are used. **Precision** is the ratio valid clones to the number of retrieved clones. In short, if $|C_+|$ is the number of valid clones identified and $|NC_+|$ is the number of invalid clones identified, then

$$precision = \frac{|C_+|}{|C_+| + |NC_+|}$$

The range is [0.0, 1.0]. Higher values mean the detected clones are more valid with fewer false positives.

Recall is the ratio of the number of accurately detected clones to the number of total actual clones. This requires a fully labeled data set. In other words, if $|C_+|$ is the number of valid clones identified and $|C_-|$ is the number of valid clones not identified, then

$$recall = \frac{|C_+|}{|C_+| + |C_-|}$$

The range is [0.0, 1.0]. Higher values mean more real clones were identified as clones with fewer false negatives.

⁴In AROMA [123], the same metric is called *Recall@k*

F1 or F-Measure, is the harmonic mean of precision and recall.

$$F1 = \frac{2 * Precision * Recall}{Precision + Recall}$$

The range is [0.0, 1.0]. Higher values mean better results.

4.8 Experimental Setup

The experiments in this study were run on a Ubuntu 18.04 LTS Virtual Machine with 32 CPUs and 64GB memory on a Dell PowerEdge R640 server with Intel Xeon Silver 4210 CPU @ 2.2 GHz and VMware ESXi 6.7.0 hypervisor to manage the VM.

Minimum token size (MIN_TOK_SIZE in section 4.2.1): Set to 3. IR based techniques [59, 110] on source code find that tokens of less than 3 characters are irrelevant.

Minimum segment size (MIN_STMTS in section 4.2.3): A small value of MIN_STMTS results in more granular snippets. MIN_STMTS is set to 1 for maximum number of behavioral snippets of code.

Maximum number of arguments (ARGS_MAX in section 4.2.3): Mathew et al. find that ARGS_MAX of 128 was sufficient for cross language clones on projects from Google Code Jam (GCJ) [145]. Although the AtCoder data is similar to GCJ [124], a more cautious approach is taken and hence set ARGS_MAX to 256.

Number of individual search results (TOP_K in section 4.2.4): This is set to 100. Hence, for each individual search based on d_{token} , d_{AST} or d_{IO} , COSAL fetches the 100 best search results.

4.9 Results

The results for each RQ are presented in turn.

4.9.1 RQ4: Single vs Multi-Objective Search

In a cross-language search context, the results of COSAL with multiple objectives are compared to COSAL with subsets of the objectives (e.g., $COSAL_{AST}$ is COSAL with only the AST objective). The validation of this study was performed using ‘leave-one-out’ cross-validation [51] where each code fragment is used as a query against all other fragments in the repository. This approach is used over the traditional k-fold cross validation since ‘leave-one-out’ is approximately unbiased and more thorough [1].

Each of the 43,146 code fragments is used as a query. The results are detailed in Table 4.5. We can see that token-based search ($COSAL_{tokens}$) and AST-based search ($COSAL_{AST}$) are less precise individually compared to dynamic search ($COSAL_{SLACC}$), but have higher recall for $k = \{5, 10\}$. When both the static similarity measures are used as parts of a bi-objective search ($COSAL_{static}$), we notice better metrics compared to each objective individually, and better metrics than the dynamic approach $COSAL_{SLACC}$ in $Precision@k$ and $SuccessRate@k$ when $k > 1$.

Table 4.5 RQ4 & RQ5: Cross-language search results on AtCoder dataset comparing COSAL against the state-of-the-practice (*SotP*) GitHub, and ElasticSearch. COSAL_{token} , COSAL_{AST} , COSAL_{SLACC} use single search similarities (*Single Sim.*) d_{token} , d_{AST} and d_{IO} respectively. COSAL_{static} uses d_{token} and d_{AST} with non-domination. $\text{KD}_{IO+AST+token}$ performs code search with KDTree using d_{token} , d_{AST} and d_{IO} . Code search techniques using multiple search similarities are represented with *Multiple Sim.*

	Search	MRR	P@1/3/5/10	R@1/3/5/10
<i>SotA</i>	ElasticSearch	29	27/25/23/24	27/44/57/75
	GitHub	37	32/36/38/39	32/49/60/73
<i>Single Sim.</i>	COSAL_{token}	31	27/31/40/42	27/48/58/72
	COSAL_{AST}	34	34/41/45/44	34/41/58/82
	COSAL_{SLACC}	45	42/42/35/27	42/45/47/47
<i>Multi Sim.</i>	COSAL_{static}	43	40/45/44/48	40/72/85/86
	$\text{KD}_{IO+AST+token}$	39	39/41/40/37	39/56/71/89
	COSAL	64	58/64/65/61	58/88/91/94

In the multiple similarity based search (COSAL), we observe the highest *Precision@k* and *SuccessRate@k* for all k as compared to the individual similarity metrics or non-dominated search with the static metrics (COSAL_{static}).

The power of the technique comes from using static and dynamic information without converting them into a single search metric. In contrast, $\text{KD}_{IO+AST+token}$ uses d_{token} , d_{AST} and d_{IO} to build a KDTree [2]. KDTree is a common approach used for information retrieval [19, 27] which aggregates d_{token} , d_{AST} and d_{IO} into a single distance metric and is eventually used to identify the nearest neighbors. Although both $\text{KD}_{IO+AST+token}$ and COSAL inherently use the same similarity measures for code search, the former has lower *MRR*, *Precision@k* and *SuccessRate@k* compared to the later. This shows that aggregation of code similarities into a single similarity does not help code search since these similarities complement each other. On the other hand, ranking the search results using non-domination of the individual similarities yields much better search results.

RQ4: Using multi-objective search with static and dynamic objectives improves the quality of results for code-to-code search compared to using one or two search objectives.

4.9.2 RQ5: State-of-the-Practice Code Search

COSAL is compared against GitHub Search (section 4.6.2) and a custom full text search based on ElasticSearch (section 4.6.1). A ‘leave-one-out’ cross-validation is adopted with each of the 43,146 code fragments as a query. Results are shown in Table 4.5.

We observe that between the textual code search tools, GitHub Search has better *MRR*, *Precision@k* and *SuccessRate@k* compared to ElasticSearch except for *SuccessRate@10*. Yet, GitHub Search and ElasticSearch are worse off compared to COSAL in all three measures. Compared to our single-objective token-based search, COSAL_{tokens} , GitHub has higher precision@{1,3} and recall at

Table 4.6 RQ6: Single-language Java code search comparing COSAL to the state-of-the-art (*SotA*) FaCoY on AtCoder and BigCloneBench.

		Search	MRR	P@1/3/5/10	SR@1/3/5/10
AtCoder	<i>SotA</i>	FaCoY	51	37/35/33/32	37/40/49/63
	<i>Single Sim.</i>	COSAL _{tokens}	46	36/32/31/29	36/40/45/58
		COSAL _{AST}	40	38/33/31/28	38/42/51/69
		COSAL _{SLACC}	40	39/39/38/32	39/48/52/59
	<i>Multi Sim.</i>	COSAL _{static}	53	43/45/44/41	43/58/65/77
		COSAL	57	50/53/54/48	50/63/75/88
BigCloneBench	<i>SotA</i>	FaCoY	76	70/68/68/65	70/72/74/81
	<i>Single Sim.</i>	COSAL _{tokens}	75	69/65/61/59	69/72/74/81
		COSAL _{AST}	72	68/61/55/51	68/74/76/83
		COSAL _{SLACC}	07	06/02/01/01	06/07/07/09
	<i>Multi Sim.</i>	COSAL _{static}	81	76/ 73/72/67	76/ 81/89/94
		COSAL	81	77/73/72/68	77/81/89/94

all levels of k . If additional static information is incorporated (COSAL_{static}), it leads to substantially better performance compared to the state-of-the-practice baselines.

RQ5: COSAL obtains better *Precision@k*, *SuccessRate@k* and *MRR* compared to GitHub Search and ElasticSearch.

4.9.3 RQ6: Single Language Code Search

FaCoY [115] is a state-of-the-art code-to-code search tool but they support only Java. Hence, COSAL is compared against FaCoY using Java code snippets only. To ensure that the dataset is not skewed due to outlier projects with limited submissions, only Java projects with 10 or more submissions are considered. This reduces the dataset to 351 problems with 20,973 Java files.

Like RQ5, a ‘leave-one-out’ cross-validation is used with each of the 20,973 code fragments as a query and the remaining problems as the search index

The results for *MRR*, *Precision@k* and *SuccessRate@k* are tabulated in Table 4.6. COSAL has better *MRR*, *Precision@k* and *SuccessRate@k* compared to FaCoY. If COSAL is used with only static similarity measures (COSAL_{static}), the *Precision@k* and *SuccessRate@k* are better than to FaCoY and COSAL_{SLACC}. Thus the use of dynamic information, aids COSAL to yield better quality search results.

Since, FaCoY supports only Java, COSAL is also compared to FaCoY using BigCloneBench. This experiment should help us evaluate the feasibility of COSAL with respect to open-source projects. Like the experiment on AtCoder, a ‘leave-one-out’ cross-validation is used where each file from BigCloneBench is used as a query and the other files are used as search results. A search result is considered valid if it has the same functionality group as the search query.

Table 4.7 RQ6: Performance of FaCoY and GitHub Search compared to COSAL based on BigCloneBench.

	Search	MRR	P@1/3/5/10	R@1/3/5/10
<i>SotA</i>	GitHub	63	58/53/52/45	58/65/67/70
	FaCoY	76	70/68/68/65	70/72/74/81
<i>Single Sim.</i>	COSAL _{tokens}	75	69/65/61/59	69/72/74/81
	COSAL _{AST}	72	68/61/55/51	68/74/76/83
	COSAL _{SLACC}	07	06/02/01/01	06/07/07/09
<i>Multi Sim.</i>	COSAL _{static}	81	76/ 73/72/67	76/ 81/89/94
	COSAL	81	77/73/72/68	77/81/89/94

Table 4.7 tabulates the mean values of *MRR*, *Precision@k* and *SuccessRate@k* for this experiment. Compared to AtCoder, the BigCloneBench dataset yields better results for all the techniques. This is due to the nature of the 43 functionalities in BigCloneBench which are distinct from each other with minimal overlap. This can be corroborated by the better scores for token based search compared to the AST based search on BigCloneBench dataset. In contrast, on AtCoder, AST based search out-performs token based search. Like the AtCoder dataset, COSAL based on individual similarity measures (COSAL_{tokens}, COSAL_{AST}, COSAL_{SLACC}) perform worse than FaCoY. But, search based on a combination of measures (COSAL_{static}, COSAL) yields better results compared to FaCoY

Only 4,984 (9%) of the files from BigCloneBench are executable by SLACC, since the remaining files depend on external libraries. As a result, dynamic similarity (COSAL_{SLACC}) has much lower scores compared to the other techniques in Table 4.7. Subsequently, the inclusion of dynamic similarity measure hardly contributes to the *MRR*, *Precision@k* and *SuccessRate@k* of COSAL as highlighted by their relatively same values for COSAL_{static} and COSAL from Table 4.7.

RQ6: Compared to state-of-the-art Java code-to-code search FaCoY, using dynamic information helps COSAL obtains better search results when executable code snippets are present. In the absence of sufficient dynamic information, using a combination of AST and token based similarities yields better results compared to FaCoY.

4.9.4 RQ7: Code clone detection

As of this study, there was no other existing tool for cross-language code-to-code search. Hence, COSAL is compared to cross-language code clone detection techniques, specifically ASTLearner, CLCDSA and SLACC. ASTLearner and CLCDSA build deep learning models and require a training, validation and testing set. Hence, the AtCoder dataset is divided into these three sets using the same approach adopted in CLCDSA [124].

Only projects with at least 20 Java and 20 Python submissions are considered. This reduces the dataset to 302 different problems. For each of the problem, ten submissions are selected each

Table 4.8 RQ7: Performance of COSAL in clone detection compared to ASTLearner, CLCDSA, and SLACC on AtCoder.

	Clone Detector	Precision	Recall	F1
<i>SotA</i>	ASTLearner	25	80	38
	CLCDSA	49	83	62
<i>Single Sim.</i>	SLACC	66	19	30
<i>Multi Sim.</i>	COSAL _{static}	48	85	61
	COSAL	55	89	68

from Java and Python as part of the training set, five for the validation set and five for the test set. The default hyper-parameters are used from ASTLearner and CLCDSA to build their models. Since COSAL and SLACC do not use machine learning models, all the submissions from the training set are added in the search database and use the testing set to evaluate COSAL and SLACC. The validation set is not considered. To account for variance, this step is repeated 10 times and report the mean precision, recall and F1 scores.

Results are shown in Table 4.8. SLACC is the most precise technique on this dataset but has extremely low *recall* compared to other techniques, and hence the lowest F1s. The low *recall* on SLACC is due to the dynamic nature of SLACC which requires valid executable code snippets (section 4.2.3). COSAL has better *precision* and *recall* compared to the static similarity based approaches ASTLearner and CLCDSA. If COSAL is used only with the static similarity measures (COSAL_{static}), the *precision* and *recall* is still better than ASTLearner and comparable to CLCDSA.

RQ7: For code clone detection, COSAL obtains better *precision*, *recall* and *F1* scores compared to ASTLearner and CLCDSA, without the need to build models. COSAL has lower *precision* to SLACC but much better *recall* and *F1* score.

4.9.5 RQ8: Code-to-code search for functional languages

In our last study the application of COSAL is validated for Haskell, a functional programming language.

4.9.5.1 Within Language

First, the results of COSAL on Haskell are compared with multiple search similarity measures to COSAL with subsets of similarity measures. For each model, the training set from Haskell is used to build the baseline models and COSAL. Like Java(Table 4.6), COSAL outperforms the other formulations that use subsets of the similarity measures. It also outperforms an alternate ranking

Table 4.9 Code search results for Haskell on AtCoder comparing COSAL against the state-of-the-practice (*SotP*) GitHub, and ElasticSearch. COSAL_{token} , COSAL_{AST} , COSAL_{SLACC} use single search similarities (*Single Sim.*) d_{token} , d_{AST} and d_{IO} respectively. COSAL_{static} uses d_{token} and d_{AST} with non-domination. $\text{KD}_{IO+AST+token}$ performs code search with KDTree using d_{token} , d_{AST} and d_{IO} . StaCE performs code search with Embedding using d_{token} and d_{AST} . Code search techniques using multiple similarity measures are represented with *Multi Sim.*

	Search	MRR	P@1/3/5/10	SR@1/3/5/10
<i>SotP</i>	ElasticSearch	29	27/25/23/24	27/44/57/75
	GitHub	31	32/32/31/31	32/49/60/76
<i>Single Sim.</i>	COSAL_{token}	31	27/31/40/42	27/48/58/72
	COSAL_{AST}	34	34/33/33/32	34/41/58/82
	COSAL_{SLACC}	45	42/42/35/27	42/45/47/47
<i>Multi Sim.</i>	COSAL_{static}	43	40/45/44/48	40/72/ 85/86
	$\text{KD}_{IO+AST+token}$	40	40/42/41/38	40/57/72/90
	COSAL	54	53/54/53/51	53/76/79/83

based on aggregating d_{token} , d_{AST} and d_{IO} to build a KDTree [2]. Although both $\text{KD}_{IO+AST+token}$ and COSAL inherently use the same similarity measures for code search, the former has lower *MRR*, *Precision@k* and *SuccessRate@k* compared to the later. This shows that aggregation of code similarities into a single similarity does not help code search since these similarities complement each other.

We can see that token-based search (COSAL_{token}) and AST-based search (COSAL_{AST}) are less precise individually compared to dynamic search (COSAL_{SLACC}), but have higher success rate for $k > 5$. Using the static similarity measures as part of non-dominated ranking (COSAL_{static}), we see better metrics compared to each similarity individually and better metrics than COSAL_{SLACC} .

When COSAL is compared against Github Search and the custom full-text search based on ElasticSearch, we observe that between the textual code search techniques, Github Search has better *MRR*, *Precision@k* and *SuccessRate@k* compared to ElasticSearch. However, both Github Search and ElasticSearch have lower values for all metrics compared to COSAL on Haskell.

4.9.5.2 Cross-Language

To validate approach on cross-language code-to-code search for Haskell, the training set from Java, Python and Haskell is used to build the baseline models and COSAL. Haskell queries from the test set are used to compute the *MRR*, *Precision@1,3,5,10* and *SuccessRate@1,3,5,10*. The median values for these metrics are reported in Table 4.10.

For all the approaches we see that the metrics for cross-language similarity is consistently lower compared to code-to-code search within Haskell (Table 4.9). We also observe that compared to the state-of-the-art and state-of-the-art techniques, COSAL has better values for *MRR*, *Precision@k* and *Recall@k*. Unlike within-language search, using IO-based similarity (COSAL_{SLACC}) is not better than AST based search similarity (COSAL_{AST}). However, we can see a large improvement in the

Table 4.10 Cross-language code search results for Haskell queries with Python and Java search results on AtCoder dataset comparing COSAL against the state-of-the-practice (*SotP*) GitHub, and ElasticSearch. COSAL_{token} , COSAL_{AST} , COSAL_{SLACC} use single search similarities (*Single Sim.*) d_{token} , d_{AST} and d_{IO} respectively. COSAL_{static} uses d_{token} and d_{AST} with non-domination. $\text{KD}_{IO+AST+token}$ performs code search with KDTree using d_{token} , d_{AST} and d_{IO} . StACE performs code search with Embedding using d_{token} and d_{AST} . Code search techniques using multiple similarity measures are represented with *Multi Sim.*

	Search	MRR	P@1/3/5/10	SR@1/3/5/10
<i>SotP</i>	ElasticSearch	23	22/23/24/23	22/28/33/50
	GitHub	25	25/26/26/21	25/32/44/57
<i>Single Sim.</i>	COSAL_{token}	25	25/25/23/23	25/30/38/51
	COSAL_{AST}	31	29/30/32/31	29/37/43/59
	COSAL_{SLACC}	30	30/30/29/26	30/33/34/37
<i>Multi Sim.</i>	COSAL_{static}	36	34/36/36/33	34/42/52/68
	$\text{KD}_{IO+AST+token}$	32	33/33/31/28	33/43/50/59
	COSAL	42	41/41/41/39	41/47/57/73

quality of the search results when we augment static similarity based COSAL (COSAL_{static}) with IO based similarity. This indicates that IO-based similarity succeeds in identifying a different set of search results compared to static similarity approaches.

Using non-dominated ranking with static and dynamic similarity measures supports code-to-code search in functional programming languages and cross-language code to code search across languages with different programming paradigms.

CHAPTER

5

STRUCTURAL AND CONTEXTUAL EMBEDDINGS

This work is submitted to ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) 2022 in collaboration with Dr. Kathryn T. Stolee [139].

5.1 Motivation

Code-to-code search requires code similarity measures that cover different developer concerns. For instance, novice programmers tend to rely on identifiers [66, 117]. Similarly, similarity measures that rely on ASTs yield high precision and recall [44, 125]. Likewise, to infer code-behavior, dynamic analysis based on IO similarity is also preferred by some developers [47, 56]. COSAL shows that individually each measure has their short comings but collectively they are greater than the sum of their parts.

Using multiple similarity measures is not very effective for large scale datasets. Fig. 5.1 compares the query times of COSAL for 100 random queries with individual search similarities (Tokens, AST, IO) on the AtCoder dataset. The models are built on varying sizes of the dataset. Token based similarity in COSAL (Tokens) indexes tokens and for a given code query looks up snippets with similar tokens within the search index. This helps in scalability and as we can see from Fig. 5.1, token based search is scalable as the dataset size increases. In contrast, the AST based search and IO based search have much larger runtimes as the size of the dataset increases. COSAL uses a squared time tree comparison [8] approach to find the similarity between ASTs. Similarly, IO based search is dependent on the execution of code snippets prior them. Additionally, non-dominated sorting

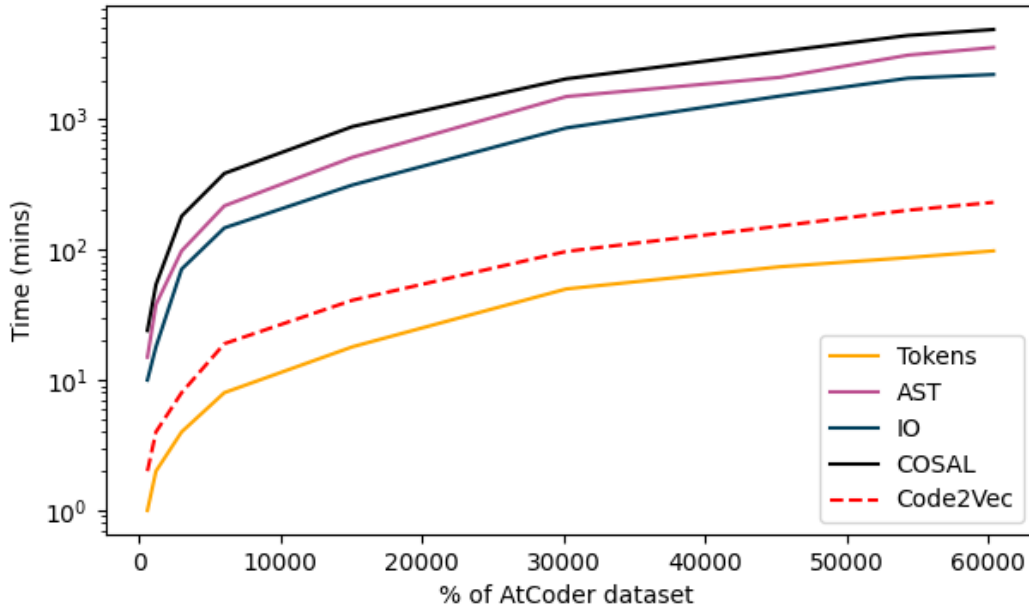


Figure 5.1 Comparing runtimes for 100 random queries comparing Code2Vec and COSAL

```

1  int fibonacci(int n) {
2      if (n <= 1) return n;
3      int prev=1, cur=1, i=2;
4      for (; i < n; i++) {
5          int t = cur;
6          cur += prev;
7          prev = t;
8      }
9      return cur;
10 }

```

(a) Java using iterative

```

1  @lru_cache(maxsize=128)
2  def fib(n):
3      if n <= 1:
4          return 1
5      return fib(n-1) + \
6          fib(n-2)

```

(b) Python using cached recursive

Figure 5.2 Java and Python implementations to get the n^{th} value from the Fibonacci series.

algorithm used by COSAL to rank code snippets across the three similarity measures has a quadratic time complexity with respect to the number of snippets. This results in the additional time incurred by COSAL compared to its individual parts and struggles to resolve ties between search results as the number of search similarity measures increase [41], which is quite common in the case of when there are more than three similarity measures [68]. In contrast, we see that Code2Vec [120], an embedding based approach based on ASTs, is an order of magnitude faster than the AST based comparison approach adopted by COSAL.

Embedding source code can infer and learn the similarity between code snippets subject to sufficient training data [114]. For example consider the of Java and Python implementations to get the n^{th} value from a Fibonacci Series in Fig. 5.2. With respect to token similarity, both the snippets do not share any common tokens. The snippets also have different structural implementations as the

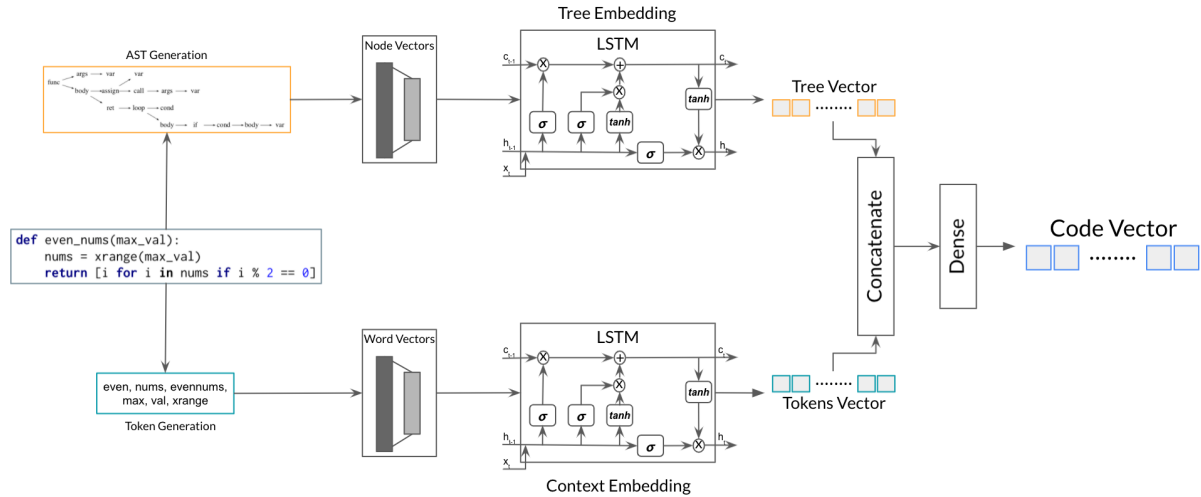


Figure 5.3 Overview of StaCE

Java variant 5.2a uses an iterative approach and the Python variant 5.2b uses a recursive approach. Lastly, 5.2b uses a decorator `lru_cache` which is not supported by COSAL which prevents IO based dynamic similarity. Hence, although the implementations Fig. 5.2a and Fig. 5.2b are functionally similar, COSAL fails to detect their similarity. However, with sufficient examples we can see that `fibonacci` and `fib` imply the same context. [108]. Similarly, the paths from iterative implementation and the recursive implementations can be trained to be similar to each other [120].

In this study, we embed structural and token based representations of code to compare code across different programming languages. The use of embedded representations of code improves the time taken to compare code snippets. The embedded representations also succeeds in identifying similar snippets of code without relying on dynamic similarity measures or common tokens and syntax trees.

5.2 Structural and Contextual Embeddings

This work presents Structural and Contextual Embeddings for Cross-Language Code Search, or StaCE for cross-language code similarity. In this section, we first describe the network architecture in a high level (section 5.2.1) followed by code preprocessing in section 5.2.2. We then delve into representing the code structure as a vector (section 5.2.3), the context in code as a vector (section 5.2.4), an aggregated representation of code as a vector (section 5.2.5), and finally training the model (section 5.2.6).

5.2.1 Network Architecture

StaCE embeds code into vectors using a deep neural network architecture. As described in Fig. 5.3, code is first converted into tokens and the generic AST. The tokens which represent context are

embedded into vectors using word2Vec [73] layer followed by a deep LSTM based context embedding layer. Similarly the generic AST is converted to vectors using a node2vec [98] layer followed by a deep LSTM based tree embedding layer. The resultant vectors are concatenated and reduced to single vector using a Dense Fully Connected neural network.

5.2.2 Pre-processing

Code snippets are first pre-processed before extracting contextual and structural features from them. File headers and imports are removed from the code snippets. This is because, developers tend to import libraries which are not used in the source code which can result in noisy tokens (section 5.2.4.1). For this study, words with non-ascii characters from are removed from comments in the code snippets. However, StaCE can be extended to support other character encodings.

StaCE supports snippets of code at file-level granularity or smaller. If snippets of code are divided across multiple files, they can be grouped into a single file after pre-processing. The data used in this study is granular at file-level and future studies will experiment on snippets of code at different granularity.

5.2.3 Vectorizing Structure

In the first stage, StaCE generates a vector which is the structural representation of the code snippet. On a high-level the code is first converted into a language agnostic AST. The nodes for the AST are pre-trained to generate their vectorized representations which are subsequently used to generate embeddings for the AST.

5.2.3.1 AST Generation

Code is first converted into a language-agnostic AST. Common language-agnostic ASTs have been few code search approaches [138, 139] to support code similarity across programming languages. InferCode uses SrcML [67] to generate common ASTs while COSAL [139] uses a custom AST by parsing code using ANTLR based grammar. Since SrcML does not support Python and Haskell, StaCE adopts custom language-agnostic AST parser adopted by COSAL.

COSAL builds a parser using a language specific adapter that maps language specific nodes into a generic AST. The parser contains a superset of language features that normalizes common control structures and normalizes variables, literals and operators. For language specific features, the adapter creates a custom node. The code snippet in Fig. 5.3 describes the AST generated for the function `even_nums`. As we can see from the tree, a `for` loop is abstracted out as a 'loop' node and the function name `xrange` is normalized to the 'call' node. The AST parser developed for COSAL was developed for Java and Python. StaCE extends the AST parser to support Haskell. A grammar for Haskell is generated using the same conventions and first generate a Haskell AST using ANTLR. StaCE uses a custom adapter like Java and Python adapters in COSAL to convert the Haskell AST to a generic AST.

5.2.3.2 Node Vectorization

This is an offline step where nodes are converted into a vectorized representation. Mou et al. [98] propose a representational learning approach to learn a program vector based on coding criteria. Although this approach was developed for C , the authors recommend this approach for any arbitrary tree based representation. Each node p is represented as a vector such that $vec(p) \in \mathbb{R}^{N_{AST}}$ where N_{AST} is the size of the embedding. For each node-leaf node p and its children c_1, \dots, c_n the $vec(p)$ is approximately represented as

$$vec(p) \approx \tanh\left(\sum_i l_i W_i \cdot vec(c_i) + b\right)$$

where $W_i \in \mathbb{R}^{N_{AST} \times N_{AST}}$ is the weight corresponding to the node c_i ; $b \in \mathbb{R}^{N_{AST}}$ is the bias and l_i is the coefficient of the node c_i which is computed as $l_i = \frac{\# \text{ leaves under } c_i}{\# \text{ leaves under } p}$. The similarity d between $vec(p)$ and its coded vector is measured by the euclidean distance d . The objective in pre-training is to minimize the similarity between $vec(p)$ and its coded vector for all the nodes.

These vector values are used as the initial weights for the Node Embedding layer which will be retrained for fine tuning as part of the larger network described in Fig. 5.3.

5.2.3.3 Tree Embedding

In this step, the encoded nodes from the generic AST is used to generate a tree vector. A standard Bidirectional LSTM [32] architecture is used with $hidden_{tree}$ hidden layers, followed by a dropout layer, a time-distributed dense layer with $hidden_{tree}/2$ hidden layers. The time-distributed results are flattened and finally sent to a dense layer with $dims_{tree}$. The final vector of with $dims_{tree}$ dimensions represents the tree vector. The input for the Bidirectional LSTM is the sequential input of the vectorized nodes from the generated AST. The nodes are inputted using the pre-order traversal of the tree to preserve the node dependencies and the small neighborhood for each node [96].

5.2.4 Vectorizing Context

Tokens from source code can be used to infer context intended by the programmer [74, 130]. In this stage, StaCE uses these tokens to create a vector which is a representation of its context.

5.2.4.1 Token Generation

Tokens are first extracted from the source code. Non-language specific tokens from source code and comments are extracted by removing language specific keywords, frequently used words in a language, common English stop words. Tokens are also split to address language-specific nomenclature like *camelCase* in Java and Haskell and *snake_case* in Python. Tokens of length less than MIN_TOK_SIZE are removed and finally converted to lower case. The tokens extracted from the code snippet `even_nums` is shown in the ‘Token Generation’ stage from Fig. 5.3.

5.2.4.2 Word Vectorization

Like embedding nodes in AST generation (section 5.2.3.2), pre-trained embeddings are generated offline for the extracted tokens and use it as the initial weights for the Word Embedding layer. StaCE uses the GloVe [83] word embeddings. In GloVe, embeddings are generated based on local context and global word co-occurrence. This makes it more suited for our application since StaCE compares occurrence of tokens across code snippets from different languages. StaCE generates initial word embeddings of size N_{tokens} for the tokens extracted from the code corpus.

5.2.4.3 Context Embedding

The embedded tokens are used to generate a context vector. The architecture of Context embedding is similar to tree embedding (section 5.2.3.3). The context embedding network uses a standard Bidirectional LSTM with $hidden_{context}$ hidden layers followed by a dropout layer, a dense layer with $hidden_{context}$. The results are then flattened and finally sent to a dense layer with $dims_{context}$ which represents the context vector. It should be noted that during context embedding, StaCE does not use a Time-Distributed layer unlike tree embedding since the order of the tokens are not relevant in inferring the context of the source [139].

5.2.5 Vectorizing Code

In the last stage, the context and structure vectors are concatenated to generate a vector of size $dims_{tree} + dims_{context}$. The concatenated vectors are finally sent through a fully connected dense layer with $hidden_{code}$ dimensions with the Rectified Linear Unit (ReLU) activation function, followed by a Dropout layer and a final fully connected dense layer of size $dims_{code}$. The vector from the final layer is normalized using L2 normalization and represents the embedded code snippets.

5.2.6 Training the model

The model architecture is trained using the triplet loss function [93] since it is more tolerant towards noisy data compared to contrastive loss. The triplet loss attempts to force similar code snippets together and push dissimilar snippets apart in the embedded space. Given an anchor code snippet a , a similar code snippet p and a dissimilar code snippet n , triplet loss tunes the model such that the distance between the embeddings of a and p is smaller than the distance between a and n . Mathematically, triplet loss minimizes

$$\max(\|e_a - e_p\| - \|e_a - e_n\| + \epsilon, 0)$$

where e_x is the code embedding for $x = \{a, n, p\}$, $\|\cdot\|$ is the euclidean distance metric, and the margin is ϵ . The margin ϵ ensures that the e_p is at least ϵ closer to e_a than e_n .

5.3 Study Design

This study poses four research questions (section 5.3.1) to evaluate StaCE on the applications of code-to-code search and clone detection using standard practice evaluation metrics and evaluate the efficiency of an embedding based approach like StaCE to COSAL, an index based method. The design for each research question is different. The data set and the baselines used to answer each research question is described in sections section 5.3.2 and section 5.3.3 respectively. The experimental setup is described in section 5.3.4 and the metrics used in evaluating COSAL and baselines are described in section 5.3.4.4.

5.3.1 Research Questions

We first evaluate the effectiveness of StaCE for code-to-code search within a programming language. StaCE experiments on Java, Python and Haskell to evaluate for languages with different programming paradigms. Hence the first research question:

Research Question 9

How effective is StaCE in code-to-code search within a programming language when compared to state-of-the-practice and state-of-the-art public code search tools?

Next, we evaluate if StaCE can be used to compare code across programming languages. Thus, our second research question:

Research Question 10

How effective is StaCE in cross-language code-to-code search compared to state-of-the-practice and state-of-the-art public code search tools?

Code-to-code search is often used in clone detection [48, 127]. Using COSAL for clone detection, we compare against ASTLearner [125], CLCDSA [124], and SLACC [134]. This leads to our last research question:

Research Question 11

Can StaCE effectively detect cross-languauge code clones?

Finally, we evaluate the practical scalability of StaCE, a model based embedded code similarity technique compared to an index based code similarity technique, COSAL. For varying sample sizes of the experimental dataset, we measure the efficiency (query time) and effectiveness (section 5.3.4.4) of StaCE and COSAL. Hence our last research question:

Table 5.1 Summaries of AtCoder (AtC) and BigCloneBench (BCB) dataset.

Metric	AtC			BCB
	Java	Python	Haskell	Java
#Problems (Functionalities)	1056	964	183	43
#Files	60,401	62,722	6,132	55,499
Avg. Files/Problem	57	65	34	1,291
#Methods	237,498	22,047	30,612	765,361
Avg. Lines/File	49	20	19	278

Research Question 12

Is StaCE more effective and efficient compared to COSAL on code repositories of varying sizes?

5.3.2 Data

This study validates search results and compares clone detection tools in cross-language and within-language environments. Like our previous studies, we reuse the same datasets AtCoder and BigCloneBench as described in Section section 4.5. However, we have expanded the Java and Python AtCoder code snippets and included snippets for Haskell.

We limit our study to the most recent 1075 problems which had solutions in Java, Python or Haskell. For these problems, we crawled 139,255 files from all the *accepted* Java, Python and Haskell solutions. Table 5.1 lists a detailed set of metrics for the study; 183 of the 1075 problems have solutions in all three languages. The data used in this study is available online [135].

5.3.3 Baselines

StaCE reuses the same baselines for the applications of code-to-code search or clone detection from COSAL as described in Section section 4.6. However, for code-to-code search StaCE is also benchmarked against Code2Vec.

Code2Vec [120] by Alon et al. represents code snippets as a fixed-length code vectors. Code2Vec first represents code as a collection of paths in the abstract syntax tree and learning the representation of each path simultaneously and aggregate them. The vectorized representations generated by Code2Vec has been used to predict method names. In our experiments we use Code2Vec vectors to compare the similarity of code snippets using euclidean distance. We also replace the Java specific AST used by authors with our language-agnostic AST approach to facilitate cross-language code similarity.

5.3.4 Experimental Setup

5.3.4.1 Hardware

Our experiments were run on a Ubuntu 18.04 LTS Virtual Machine with 32 CPUs and 64GB memory. We use a Dell PowerEdge R640 server with Intel Xeon Silver 4210 CPU @ 2.2 GHz and VMware ESXi 6.7.0 hypervisor to manage the VM.

5.3.4.2 Validation

Before building the baseline models, the data is split into training, validation, and testing sets in the ratio 80%, 10%, and 10% respectively. Though this approach directly does not correspond to the real task of code-to-code search, it has been widely used for training similar models [120, 122, 139]. To ensure that the results are not random, we repeat this approach 10 times using a 10-fold cross validation approach after randomly shuffling the data. The median value for each of the evaluation metrics (section 5.3.4.4) from the all the folds are reported.

5.3.4.3 Hyperparameters

N_{AST} in section 5.2.3 is set to 300. $hidden_{tree}$ and $dims_{tree}$ in section 5.2.4 is set to 256 and 64 respectively. Like COSAL, we set MIN_TOK_SIZE to 3. IR based techniques [59, 110] on source code find that tokens of less than 3 characters are irrelevant. In section 5.2.4, the GloVe word embeddings (N_{tokens}) is set to 256, $hidden_{context}$ is set to 256 and $dims_{tree}$ is set to 64. For the baseline models, we reuse the same parameters suggested by the respective authors.

5.3.4.4 Evaluation metrics

StaCE reuses the same code-to-code search metrics and clone detection metrics as COSAL. StaCE uses $Precision@k$, $SuccessRate@k$, and MRR (section 4.7.1 in the context of code-to-code search. For clone detection, StaCE uses $Precision$, $Recall$ and $F1$ score (section 4.7.2).

5.4 Results

In this section we answer each of the research questions posed in turn.

5.4.1 RQ9: Within language code-to-code search

In our first study, we compare StaCE in the application of code-to-code search within a programming language. We compare StaCE with ElasticSearch, Github Search, Code2Vec, COSAL and COSAL with static similarity measures. StaCE is evaluated on two datasets: AtCoder and BigCloneBench. Since the AtCoder dataset contains snippets of similar code across Java, Python and Haskell, it is used to evaluate the application of StaCE on different programming languages. BigCloneBench is

Table 5.2 RQ9: Within-language search results in Haskell, Java and Python on AtCoder dataset and Java on BigCloneBench dataset. StaCE is compared against the state-of-the-practice (*SotP*) GitHub and ElasticSearch and state-of-the-art (*SotA*) Code2Vec [120] and COSAL. The table reports the MRR (M), Precision@1,3,5,10 (Prec) and SuccessRate@1,3,5,10 (SRate) for the queries and search results in the same language.

Search	AtCoder								
	Haskell			Java			Python		
	M	Prec	SRate	M	Prec	SRate	M	Prec	SRate
ElasticSearch	28	28/28/29/28	29/33/42/55	34	32/30/29/29	32/48/61/77	33	33/32/31/31	33/51/62/73
GitHub	31	30/30/29/28	30/35/51/62	39	39/38/37/38	39/54/65/74	40	40/40/39/39	40/54/65/77
Code2Vec	43	43/42/42/41	43/49/58/67	58	58/58/56/55	58/68/82/92	57	57/56/56/55	57/69/80/89
COSAL	51	51/51/50/49	51/57/66/74	66	66/65/63/63	66/85/87/93	68	68/67/67/64	68/87/90/94
COSAL _{static}	49	49/48/48/46	49/56/63/73	61	62/60/60/69	62/85/87/93	62	62/62/61/60	62/85/89/92
StaCE	53	53/52/52/49	53/58/68/79	68	68/67/66/63	68/86/90/95	69	70/69/68/68	70/89/93/94

Search	BigCloneBench (Java)		
	M	Prec	SRate
ElasticSearch	53	53/53/52/52	53/62/69/75
GitHub	58	59/58/57/57	59/65/72/77
Code2Vec	71	72/71/70/65	71/74/77/81
COSAL	75	76/73/73/68	76/81/89/94
COSAL _{static}	75	75/73/72/67	75/81/89/94
StaCE	79	79/78/78/75	79/83/92/96

a collection of similar Java snippets harnessed from Github. Hence, it is used to demonstrate the application of StaCE on practical real-world code snippets.

We use the training set for each language in both the datasets to build the baseline models and StaCE. The snippets from the test set in each language are used as queries and Table 4.6 reports the median of Precision@1,3,5,10, Recall@1,3,5,10 and MRR.

With respect to the state-of-the-practice tools, StaCE clearly outperforms both ElasticSearch and Github across all three languages. Code2Vec [120] is also a model based technique that embeds source code into vectors which can be used to compare them. Like StaCE, it is much better than state-of-the-practice tools but the StaCE outperforms it. Unlike Code2Vec which uses paths extracted from the AST to embed code, StaCE uses contextual information extracted from tokens and structural information extracted from ASTs to embed code into vectors. The effectiveness of multiple representations to compare code code is also highlighted by the higher scores of COSAL compared to Code2Vec. COSAL uses two static similarity measures based on tokens and AST and a dynamic similarity measure based on IO behavior to compare source code.

Although StaCE outperforms COSAL for all the three programming languages, the gains are marginal. However, the success of COSAL can be attributed to the dynamic IO behavior. When, COSAL is used without the dynamic similarity measure (COSAL_{static}) on the AtCoder dataset, we see that it is not as effective as StaCE. The drop-off is particularly evident in the case of Python (82%) and Java (68%) with more executable code compared to Haskell (54%). This is further reinforced by

the similar performance between COSAL and COSAL_{static} on the BigCloneBench dataset. In this dataset only 8% of the code is executable and hence the the contribution from IO based similarity is minimal. We dive deeper into the role of dynamic similarity in section 6.3.3. Finally, we also see relatively higher scores for StaCE compared to COSAL and COSAL_{static} on the BigCloneBench dataset. This shows that on large real-world public code, an embedded approach is more effective compared to an index based approach.

With respect to Precision@k, SuccessRate@k and MRR, across Java, Python and Haskell StaCE is a) better than state-of-the-practice and embedding based code search tools; b) comparable to index based approaches using static and dynamic measure; c) better than index based approaches using static measures; and d) more effective than all other approaches on real-world public code.

5.4.2 RQ10: Cross-language code-to-code search

Table 5.3 RQ10: Cross-language search results on AtCoder dataset in Haskell, Java and Python comparing COSAL and StaCE against the state-of-the-practice (*SotP*) GitHub and ElasticSearch and state-of-the-art (*SotA*) Code2Vec [120]. Queries are in each language and search results in the other two languages. COSAL_{token} , COSAL_{AST} , COSAL_{SLACC} use single search similarities (*Single Sim.*) d_{token} , d_{AST} and d_{IO} respectively. COSAL_{static} uses d_{token} and d_{AST} with non-domination. StaCE_{tokens} and StaCE_{AST} uses the Tokens Vector and Tree Vector from Fig. 5.3. Techniques using multiple similarity measures are represented with *Multi Sim.*

Search	Haskell			Java			Python		
	M	Prec	SRate	M	Prec	SRate	M	Prec	SRate
ElasticSearch	23	22/23/24/23	22/28/33/50	29	27/25/23/24	27/44/57/75	30	30/30/29/26	30/47/59/72
GitHub	25	25/26/26/21	25/32/44/57	35	33/34/34/32	33/49/61/73	37	36/37/38/37	36/51/62/71
Code2Vec	35	34/35/34/32	34/40/49/64	40	38/41/41/40	38/65/78/80	43	42/45/46/45	42/68/75/78
COSAL	42	41/41/41/39	41/47/57/73	62	63/62/62/59	63/86/89/92	65	65/65/66/63	65/89/91/94
COSAL_{token}	25	25/25/23/23	25/30/38/51	28	26/28/37/39	26/45/55/69	34	30/34/43/45	30/51/61/75
StaCE_{tokens}	28	29/28/26/25	29/33/43/55	39	39/41/45/44	39/46/63/82	42	41/44/48/47	41/49/66/85
COSAL_{AST}	31	29/30/32/31	29/37/43/59	31	31/32/32/31	31/38/55/79	37	37/36/35/35	37/44/61/85
StaCE_{AST}	34	34/33/32/29	34/40/49/61	45	43/45/45/43	43/57/67/88	48	46/48/48/46	46/58/70/91
COSAL_{static}	36	34/36/36/33	34/42/52/68	40	37/42/41/45	37/69/82/83	45	42/48/47/47	42/75/86/88
StaCE	43	42/41/40/37	42/49/61/77	61	60/61/61/62	60/84/87/90	65	64/67/68/67	64/89/92/95
StaCE+IO	41	41/41/39/38	41/48/61/75	61	60/61/61/59	60/84/86/90	63	63/63/62/61	63/89/91/94

COSAL demonstrates the merits of by using multiple similarity measures using non-dominated sort. However, as discussed in section 5.2, COSAL suffers from scalability issues.

Like RQ1, we compare StaCE to ElasticSearch, Github Search, Code2Vec and COSAL. Additionally, we also use subsets of COSAL using the token-based (COSAL_{token}) and AST-based (COSAL_{AST}) similarity measures exclusively. We also use two variants of StaCE that use only the tree vector

(StACE_{AST}) and the tokens vector (StACE_{tokens}). To build these variants, we use the same approach as in section 5.2.6 but limit it to the tree vector generated by Tree Embedding and tokens vector generated by Context Embedding for StACE_{AST} and StACE_{tokens} respectively.

For each model, we use the training set from Java, Python and Haskell to build the baseline models, COSAL and StaCE. The results are detailed in Table 5.3 where we present the MRR, Precision@1,3,5,10 and SuccessRate@1,3,5,10 when each of Java, Python and Haskell snippets from test set are used as a search query to look for across languages from the training set.

We see that using the embedded representations of token (StACE_{tokens}) and AST (StACE_{AST}) outperform the direct comparison approach adopted by COSAL_{token} and COSAL_{AST} respectively. In COSAL, token similarity was computed using the Jaccard similarity of extracted tokens from the code snippets. This approach fails to capture the semantics of tokens within their context which can be addressed by the Word Embedding and Context Embedding layer in StaCE. Similarly, the Zhang-Sasha edit distance used to compare similarity between ASTs fail to capture the semantic similarity between sequences of nodes. The Node Embedding and Tree Embedding layer in StaCE overcomes this by generating similar vectors for similar sequence of nodes. This shows that the vectorized representations of trees and tokens is a better representation for comparing code.

COSAL and StaCE outperform the state of the practice tools and Code2Vec a state of the art tool for all three similarity approaches. However, we see that Code2Vec which is also an AST based approach outperforms COSAL_{AST} but does not perform better than StACE_{AST} . This further strengthens the argument for an embedding based approach to compare ASTs. However, using a branch based embedding approach like StaCE is more effective than a path sequence based embedding approach like Code2Vec.

Lastly, we notice that COSAL and StaCE are relatively similar to each other across the three languages. More specifically StaCE marginally outperforms COSAL in case of Haskell, COSAL marginally outperforms StaCE in case of Java and they are relatively same in case of Python. However, it should be noted that StaCE uses only two static similarity measures (tokens and AST) while COSAL uses a dynamic IO based similarity measure in addition to the static similarity measures. The effectiveness of dynamic similarity measures is predicated on the presence of standalone executable code which is scarce in open-source projects [56, 101, 134] which renders COSAL less viable and not unscalable. When COSAL is limited to its static similarity measures (COSAL_{static}), we notice that StaCE outperforms it across all three languages. Thus, the model used in StaCE is a viable alternative for the dynamic IO based similarity approach in COSAL.

StaCE obtains better *Precision@k*, *SuccessRate@k* and *MRR* compared to state-of-the-practice tools GitHub Search and ElasticSearch. Across Java, Python and Haskell, compared to state-of-the-art code-to-code search technique Code2Vec, COSAL obtains better search results. However, StaCE is not dependent on dynamic information and obtains similar results to COSAL using a combination of AST and token-based features.

5.4.3 RQ11: StaCE for clone detection

Although code-to-code search is a part of clone detection, they are inherently different. For a given code snippet, code clone detection returns an identical code snippet and code-to-code search returns a set of potentially relevant snippets. We compare StaCE to other state of the art clone detection techniques: ASTLearner, CLCDSA, SLACC and COSAL. COSAL inherently a code-to-code search tool, selects the top-1 ranked search result returned by non-dominated ranking. In this study, we limit our dataset to Java and Python since ASTLearner and CLCDSA support only Java and Python and does not provide a framework that can be expanded to a functional programming language like Haskell.

Like StaCE, ASTLearner and CLCDSA build deep learning models and require a training, validation and testing set. Hence, we use an experimental setup similar to the approach followed by CLCDSA [124]. We only consider projects with atleast 20 Java and 20 Python submissions, reducing the dataset to 891 different problems. For each problem, we five submissions each from Java and Python as part of the test set and five submissions as part of the validation set. The rest of the submissions from the problem are used as the training set. We use the default hyper-parameters from ASTLearner and CLCDSA to build their models. Since COSAL and SLACC do not use machine learning models, we add all the submissions from the training set to the search database and use the test set for evaluation. We do not include the validation set in the search database to ensure a fair comparison to ASTLearner and CLCDSA. Like our prior experiments, we repeat this step 10 times to account for variance and report the median values of Precision, Recall and Variance.

The results for this study are shown in Table 5.4. The techniques that use a single similarity measure (*Single Sim.*) from those that use multiple similarity measures (*Multi Sim.*). SLACC is the most precise approach for this dataset but has low recall compared to other approaches, which results in the low F1. The low recall on SLACC is overcome by the static similarity approaches as seen by the high recall scores of COSAL. Using StaCE with the static similarity approaches performs better than COSAL with respect to precision. This removes the need for IO based similarity which is slow and has a low recall. However, using the static similarity approaches with non-dominated ranking ($COSAL_{static}$) does not perform as well as StaCE. This highlights the effectiveness of the embedding technique compared to non-dominated ranking.

Across Java, Python and Haskell, StaCE obtains better *precision* and *recall* compared to code-clone detection tools based on static similarity measures like ASTLearner and CLCDSA. Compared to SLACC, a dynamic code clone detection tool, StaCE has better *recall* and *F1-score* and similar precision.

5.4.4 RQ12: Exploring Scalability

In section 5.4.1, section 5.4.2 and section 5.4.3, we see that StaCE and COSAL have similar code-to-code search and code clone similarity metrics across all three languages. However, since the StaCE

Table 5.4 RQ11: Cross-language performance of StaCE in clone detection compared to ASTLearner, CLCDSA, SLACC and COSAL on AtCoder for Java and Python.

	Clone Detector	Precision	Recall	F1
<i>Single Sim.</i>	ASTLearner	32	79	46
	CLCDSA	53	82	64
	SLACC	62	21	31
<i>Multi Sim.</i>	COSAL _{static}	54	85	66
	COSAL	59	87	70
	StaCE	61	87	72

Table 5.5 Comparing StaCE and COSAL on varying sizes of {1,2,5,10,20,50,100}% of the AtCoder dataset.

Size	Search	Haskell			Java			Python		
		M	Prec	SRate	M	Prec	SRate	M	Prec	SRate
1%	StaCE	07	07/08/07/05	07/10/14/15	11	10/11/09/07	10/14/17/19	15	14/12/09/08	14/15/16/19
	COSAL	37	37/36/35/30	37/44/53/68	51	51/49/48/44	51/57/61/67	54	54/51/51/49	54/57/60/64
2%	StaCE	06	06/07/05/05	06/10/13/14	11	11/10/09/09	11/15/18/20	14	15/14/12/11	15/18/20/21
	COSAL	38	38/38/35/33	38/45/56/71	54	54/52/51/48	54/67/74/81	57	58/56/55/53	58/72/80/84
5%	StaCE	10	09/10/09/09	09/14/17/20	15	15/15/13/12	15/19/20/22	17	16/16/15/11	16/21/23/25
	COSAL	37	38/37/35/34	38/46/49/69	55	55/55/53/50	55/70/78/83	59	58/57/56/53	58/74/81/85
10%	StaCE	14	15/14/10/09	15/21/24/30	22	22/21/19/19	22/25/27/30	24	25/24/24/21	25/29/33/36
	COSAL	40	39/39/38/35	40/46/51/72	57	58/57/57/52	58/73/81/85	59	59/58/57/52	59/78/82/87
25%	StaCE	17	18/18/17/15	18/25/30/31	25	26/25/24/24	26/29/32/35	28	28/28/27/26	28/32/34/38
	COSAL	40	39/39/38/36	40/46/52/72	59	59/58/58/52	59/76/83/87	59	59/59/57/54	59/80/83/89
50%	StaCE	38	38/37/37/36	38/46/55/70	56	56/55/55/53	56/68/76/83	52	52/52/52/48	52/64/74/85
	COSAL	42	42/42/41/38	42/45/55/70	60	60/62/61/59	60/81/87/90	64	64/64/63/62	64/83/87/90
75	StaCE	43	43/43/41/38	43/48/58/74	60	61/60/59/59	61/84/87/89	55	55/57/57/55	55/76/81/89
	COSAL	43	43/43/42/38	43/46/56/71	61	61/63/62/59	62/82/88/90	65	64/64/65/62	64/84/88/90
90%	StaCE	44	44/43/41/37	44/49/60/75	61	61/61/60/60	61/85/87/89	63	64/63/63/60	64/89/92/94
	COSAL	43	43/42/41/37	43/44/57/73	63	62/62/61/60	62/85/87/92	64	65/64/64/63	65/88/91/92
100%	StaCE	43	42/41/40/37	42/49/61/77	61	60/61/61/62	60/84/87/90	65	64/67/68/67	64/89/92/95
	COSAL	42	41/41/41/39	41/47/57/73	62	63/62/62/59	63/86/89/92	65	65/65/66/63	65/89/91/94

uses a deep learning model the size of data used to train the model, can impact the effectiveness of this StaCE. Similarly, the size of data can impact the search experience in COSAL due to the varying time complexities of the individual search similarities. In this section, we explore the impact of the size of data on COSAL and StaCE.

Table 5.5 shows the impact on varying data sizes for code-to-code search on StaCE and COSAL. We use 1%, 2%, 5%, 10%, 25%, 50%, 75%, 90% and 100% of the AtCoder data for both the approaches. For each dataset, we use the approach detailed in section 5.2.6 and report the median values of MRR, Precision@1,3,5,10 and Recall@1,3,5,10. We notice that the metrics for COSAL on Haskell stay relatively similar as the data size increases from 1% to 100%. On Java and Python queries, there is a

Table 5.6 R12: Comparing training and query time in minutes for StaCE and COSAL varying data sizes for {1,2,5,10,20,50,100}% of data. The training time refers to the time taken to build the prediction model and the query time is the time taken to search 1000 random queries.

Size	Search	Haskell		Java		Python	
		Train	Query	Train	Query	Train	Query
1%	StaCE	16	2	19	2	18	2
	COSAL	-	17	-	18	-	17
2%	StaCE	30	4	34	5	31	4
	COSAL	-	41	-	44	-	42
5%	StaCE	85	10	105	12	97	11
	COSAL	-	105	-	121	-	118
10%	StaCE	150	19	177	22	163	21
	COSAL	-	235	-	285	-	267
25%	StaCE	311	50	348	56	326	54
	COSAL	-	649	-	781	-	712
50%	StaCE	594	99	712	112	683	108
	COSAL	-	1561	-	1847	-	1697
75%	StaCE	891	154	1003	171	977	162
	COSAL	-	2475	-	2912	-	2801
90%	StaCE	1019	191	1208	211	1161	201
	COSAL	-	3318	-	4108	-	3812
100%	StaCE	1095	203	1293	223	1225	211
	COSAL	-	3714	-	4396	-	4061

more gradual improvement on all the search similarity metrics. However, StaCE has worse search similarity measures for smaller dataset sizes on Java, Python and Haskell queries. We notice that until we use at least 50% of the dataset size the queries from all three languages yield poor results. As the dataset size increases beyond 75%, there is not a large increase for similarity measures on Haskell and Java search queries compared to using the entire AtCoder dataset. For the Python search queries, we observe a similar trend at 90% of the AtCoder dataset. The large dependence for building deep learning models is a potential bottleneck for code-to-code search on smaller repositories and has led to semi-supervised learning approaches [110, 115].

This leads us to believe that COSAL is strictly better than StaCE due to the stable similarity metrics across varying dataset sizes but the search experience varies due to the large runtime for COSAL. COSAL has no training time since it does not build a machine learning model. Rather, COSAL indexes different representations (token, AST, IO) of code and subsequently looks them up while querying. StaCE first builds the deep learning model (Fig. 5.3) called training. StaCE also indexes the vectorized representations which is then used to compare similar search queries. Table 5.6 lists the training and query time in minutes for StaCE and COSAL when the dataset size is varied from 1%, 2%, 5%, 10%, 25%, 50%, 75%, 90% and 100%. Training time (*Train*) is the time taken to build the prediction model and query time is the time taken to search random queries. Although COSAL does not have a training time, the query time increases drastically as the size of the dataset increases due to the time taken to execute snippets of code for behavioral similarity (d_{IO}) and also compare ASTs for structural similarity (d_{AST}). StaCE has a one-time training time that scales linearly with the size of the training data. The query time for StaCE also increases with the size of data but the rate of increase is much more gradual compared to COSAL. This shows that COSAL has issue scaling with large data repositories.

StaCE is effective for larger datasets and is more scalable for larger datasets with better query times compared to index-based approaches. For smaller datasets, COSAL has better MRR compared to StaCE.

5.4.5 Threats to Validity

Language Bias. COSAL and StaCE were implemented for Java, Python and Haskell since the state-of-the-art code-to-code search and clone detection techniques support these languages. For other languages the results may vary depending on their complexity, syntactic abstraction, typing scheme and dynamic execution. There will also be features that are specific to a programming language and hence the generic AST will need to be extended to support it.

Baseline Bias The ElasticSearch baseline for cross-language code-to-code search is not an exact representation of a code-to-code search tool used by developers [92].

Data Bias The data are from a programming contest, which is not representative of industrial or open source coding practices. AtCoder was used since it has a vetted set of similar code snippets used in code search and clone detection studies [124, 125]. However, StaCE for open-source code

will be explored in future studies. The code snippets used in this study were at a file-level granularity. StaCE can be practically used at a smaller granularities of code. We leave exploration of different code granularities for future studies.

Similarity Bias StaCE uses two similarity measures based on syntactic features for code search based on context and structure. Other similarity measures [20, 79] are not explored in this study and we leave it for future studies.

Hyperparameter Bias StaCE uses multiple hyperparameters which we detail in section 5.3.4. For Node Vectorization and GloVe we reuse the hyper-parameters recommended by the authors. For the other components of COSAL, we use engineering judgement and grid-search to optimize $hidden_{context}$, $hidden_{tree}$, $dims_{context}$ and $dims_{tree}$.

Model Selection. We have seen the rise of advanced deep learning models to vectorize text and source code. More sophisticated embedding techniques can be used based on transformers [113] which we leave for future work.

CHAPTER

6

DISCUSSION

Chapters chapter 3, chapter 4 and chapter 5 presents three different approaches that build on each other to compare code snippets. This chapter, discusses extensions, applications and facets of each of the these techniques in turn. section 6.1 discusses the nature of clone with respect to input sizes, types of arguments and size of code. section 6.2 discusses the scalability of COSAL to open-source projects and presents how COSAL can be extended to support other languages and new similarity measures. section 6.3 analyses the nature of results generated by embedded similarity, explainability of code and merits and demerits of embedding dynamic similarity of code.

6.1 Dynamic code clone detection

We have demonstrated how SLACC can successfully identify clones in single-language, multi-language, static typed language, and dynamic typed language environments. Compared to prior art (HitoshiIO), SLACC identifies a superset of the clusters and with higher precision. Compared to type-III clone detection, SLACC achieves a much higher precision in Python and in cross-language situations. This would lead us to believe that traditional methods that detect syntactic type-III clones cannot be used for cross-language clone detection, despite successful applications in single languages for identifying libraries with reusable code [24], detecting malicious code [40], catching plagiarism [14] and identifying opportunities for refactoring [82].

Next, we explore the sensitivity of code clones to the number of inputs, the number of arguments, and the size of the snippets.

Table 6.1 Mean and variance (in parenthesis) of # clones, # clusters and # false positives for 20 repeats when # inputs varying between 8-256. The mean (and variance) are reported.

# Inputs	# Clones	# Clusters	# False Positives
8	4461(85)	218(16)	184(19)
16	4297(49)	355(17)	142(19)
32	4221(23)	412(13)	101(5)
64	4194(4)	623(6)	71(3)
128	4180(0)	630(1)	52(0)
256	4180(0)	632(0)	50(0)

6.1.1 Impact of input sizes

Prior studies have shown that varying the number of inputs can alter the accuracy of clone detection techniques [47, 54, 104]. This was particularly evident in the earliest clone detection techniques by Jiang and Su [47] where the authors limited the number of inputs to 10 with a maximum of 120 permutations of the input due to the need for large computational resources and the corresponding runtime.

We test the impact on clones, clusters, and false positives by varying the number of inputs from 8 to 256 in powers of 2 and repeating SLACC using the generated Java functions. Each experiment is repeated 20 times on a set of randomly generated inputs. For each set of input, we record the mean and variance for the number of clones, clusters and false positives, as shown in Table 6.1. For a given number of inputs, each row represents the mean and variance (in parenthesis) of the number of clones, clusters and false positives. For low numbers of inputs, we see more functions being marked as clones and fewer clusters. As the number of inputs increases, the number of clones reduces and the number of clusters increases, demonstrating that the additional inputs are critical at differentiating behavior between functions. The counts of clones, clusters, and false positives appear to plateau after 64 inputs. This highlights that 10 inputs used by Jiang and Su would not be sufficient for optimally identifying true functional clones and will lead to a large number of false positives, as suggested in prior work [56].

6.1.2 Types of clones

Table 2.1 broadly describes the 4 types of code clones. Type-I clones and type-II clones are almost exact matches and are few in number (261). But from Table 3.3 we can see that type-III clones are much more in number (10709). Of these 10709 clusters, only 432 clusters are valid semantic clones. This is because, type-III clones represent snippets of functions with small variations in their syntax which influence their behavior. Type-IV clones generated by SLACC are lesser in number compared to type-III clones. That said, they are more precise with respect to behavioral equivalence.

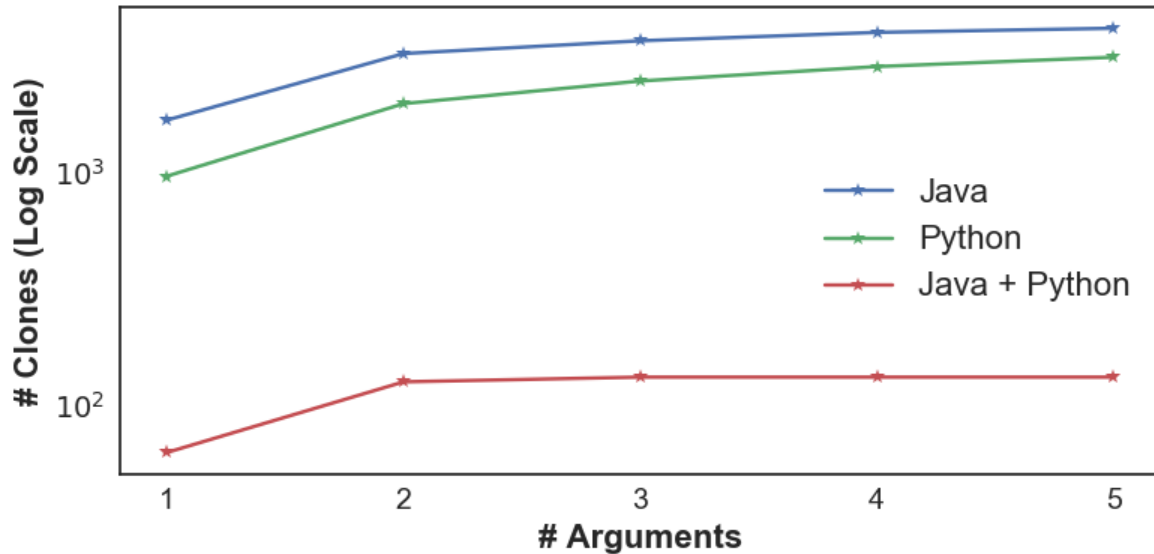


Figure 6.1 Cumulative # clones with # arguments varying between 1-5.

6.1.3 Influence of arguments in clones

We use our engineering judgment to set `ARGS_MAX = 5` (Maximum number of Arguments) to limit the number of functions generated from snippets. Fig. 6.1 represents the cumulative number of clones with arguments varying from 1 to 5 and can be used to justify our choice of `ARGS_MAX`. Most clones detected by SLACC have two arguments or less. In Java functions, 3252 of 4180 clones detected have less than three arguments. Cross-language functions are fewer in number and typically contain functions with 2 arguments or less (125 out of 131). This would seem intuitive as modular functions are more frequent compared to complex functionalities. As `ARGS_MAX` increases, it begins to plateau around 3. Hence, a larger value of `ARGS_MAX` may not yield significantly larger number of code clones but would incur more computational resources (`ARGS_MAX!` function executions).

6.1.4 Clones vs Lines Of Code

Prior work suggests there is more code redundancy at smaller levels of granularity [102]. Aggregating all the cloned functions identified by SLACC in RQ1, RQ2, and RQ3, we have 6,536 total, valid cloned functions in Java and Python (duplicates removed, as the same function could be included in an RQ1 and an RQ3 cluster, for example).

Fig. 6.2 represents the number of clones with lines of code varying from 1 to 29. Clones with 30 or more lines are denoted as “30+”. More than 50% of the valid Java clones have 6 lines of code or less (2037/3845), while the median of valid Python clones have 5 lines or less (1372/2691). This implies that snippets with more lines of code are more unique and harder to clone functionally. On the contrary, smaller snippets are more likely to contain clones in a code base. The greater median for Java clones compared to Python clones can be attributed to the verbosity in Java compared to

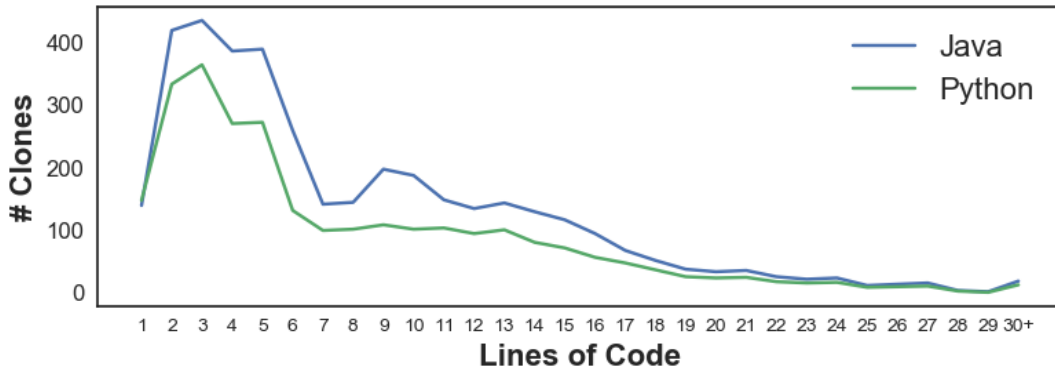


Figure 6.2 # clones for lines of code between ranging from 1-29. Clones with 30 or more lines are grouped into 30+

the succinct nature of Python [28].

6.1.5 Extending to support other languages

Through SLACC we show that using segmentation of source code and generating arguments based on grey-box testing, clones can be detected across Java and Python. This shows that cross language cloned detection is possible and could be extended to support other languages. Though this process is not trivial, it can be achieved by following a systematic set of procedures described in our repository. Although the argument generation and clustering phases are common, phases like segmentation, transformation and execution are based on a common language independent API. This API must be implemented for each of the specific phases to support a new programming language. In this study, we use Apache Thrift¹ which helps define data types and service interfaces in a simple definition file. Taking that file as input, the compiler generates code to be used to easily build Remote Procedure Call (RPC) clients and servers that communicate seamlessly across programming languages.

Cross language clone detection requires an additional configuration file describing the mappings between the target languages. Currently, we use a JSON file which contains mappings between datatypes of the target languages and the bounds for each of these types. We believe that using and evolving the guidelines in the repository and subsequent open-sourcing can help SLACC cater to a large set of programming languages.

6.1.6 Applications

Although the focus of our study was identifying semantic code clones across programming languages, we envision additional research applications for SLACC

¹thrift.apache.org

6.1.6.1 Code Translation

Since SLACC was able to detect code clones across programming languages, it could be extended to a semantic source code translator. Existing migration tools exist help translate code from one specific language to another [144] cannot be leveraged to dynamic typed languages; while more generic approaches are based on deep-learning [108] or graph matching [23] which are computationally intense and requires a large code base to develop a sustainable model. SLACC identifies equivalence for the smallest of behaviors and helps map analogous relationships between code snippets agnostic to the underlying language. This could incur an initial cost in building such an extensive data-store to store all the IO relations for snippet-functions, but subsequent queries for code translations could be made fast by utilizing a read-heavy database [64].

6.1.6.2 Automatic Program Repair

Semantic code search has been used in automatic code search and repair applications [53, 63, 74, 85, 89]. SLACC can thus be utilized for automatic program repair that is based on the availability of large code-repositories of redundant code [80]. SLACC has to be modified to store the code-clone clusters using an efficient data structure like a graph database [65, 118] that would enable search based on IO behavior. The practical implementation of a subsequent repair operator would be a good research opportunity.

6.1.6.3 Learning Programming

Research shows that the best way to learn a programming language is via examples [33, 84]. SLACC can be used as an automated tool to help students work with examples of programming language. A student can look up examples for a problem multiple ways. First, when sample inputs and outputs are provided, SLACC could return all the examples that satisfy the IOs. Second, in case of an alternate implementation, it could be transformed, executed and using the subsequent IOs clones could be looked up in the datastore. Finally, the student could provide an example in an alternate language. In such a case if the language is supported by SLACC and the type mappings exist between the languages, SLACC could be applied to find a clone.

6.1.6.4 Learning Transfer

Learning transfer² which is an extended application of code translation could be applied in contexts, such as code review tools, and within integrated development environments (IDE) such as Eclipse and Visual Studio. Learning transfer provides an added functionality of explaining how the translation from the source to the target language along with explanations in terms of support to aid the learning process. For example, consider the scenario of code translation (section 6.1.6.1) where

²Importantly, we point out that *learning transfer* is quite different than *transfer learning*. The former relates to human transfer of knowledge whereas the latter describes when machine learning algorithms use previously learned knowledge for a new domain [6].

code needs to be translated from one programming language to another: this activity is an instance in which learning transfer is required. Existing tools assist programmers by providing explanations in terms of their expert language through existing affordances in development environments. Since SLACC works at fine granularity of code, an explanation could be provided bottom-up by describing the translation using the IO relations as examples. This process would require additional steps like vetting out IOs that are hard to comprehend and improving the tool to support feedback based updates.

Learning transfer tools can be beneficial even when the language conversion is automatic. For example, SMOP (Small Matlab and Octave to Python compiler) is one example of a transpiler—the system takes in Matlab/Octave code and outputs Python code.³

6.2 Cross-language code similarity using non-dominated ranking

We have evaluated COSAL against single objective code-to-code search, state-of-the-practice code-to-code search, state-of-the-art within-language code-to-code search, and state-of-the-art cross-language code-to-code clone detection. In all cases, it outperforms the competition. Furthermore, we do not need to build, train, or update models in using COSAL. Future work will evaluate COSAL in the applications suggested in this work, such as program repair, code translation, and refactoring. In this section, we discuss the potential for scalability in adding new languages and new objectives, as well as threats to the validity of our work.

6.2.1 Scalability and Open-Source Support

To validate our experiments on the merit of using multiple search similarities for code search, COSAL was validated on AtCoder and BigCloneBench. These datasets were used to benchmark many of the prior state of the art code search and clone detection techniques [99, 101, 102, 115]. That said, both these datasets have their limitations. Firstly, since the AtCoder dataset is based of submissions from a programming contest, it is not a true representation of open-source code. Secondly, the BigCloneBench dataset has distinct functionalities which represent code clones. This makes the clone detection and code search relatively easier. This can be inferred by the relatively higher scores for all the search similarities for BigCloneBench (Table 4.7) compared to AtCoder (Table 4.6). To study the scalability of COSAL on arbitrary open source projects, we consider three popular open-source libraries for Java and Python: a) Guava Java library by Google; b) `commons-collections` Java library by Apache Software Foundation; and c) `collections` Python 2.7 system library.

Fig. 6.3 contains five specific code samples identified by COSAL on these three open-source libraries. Consider the code snippet Fig. 6.3a which counts the number of occurrences of an object in the `MultiSet`. A `MultiSet` is a special type of a `Set` which allows for multiple instances for each of its objects. When COSAL is used to look up similar code snippets to Fig. 6.3a, we notice different code snippets based on the choice of similarity. Based on tokens extracted from Fig. 6.3a, Fig. 6.3b was

³github.com/victorlei/smop

```

1 class HashMultiSet<E> ... {
2     ...
3     public int count(Object element) {
4         Count frequency = Maps.safeGet(backingMap, element);
5         return (frequency == null) ? 0 : frequency.get();
6     }
7     ...
8 }

```

(a) Method that returns count of a MultiSet from google-guava

```

1 class IterableUtils {
2     ...
3     /**
4     * ... element ... iterable ... obj ... search ... cardinality ...
5     */
6     static <E, T> int frequency(Iterable<E> iterable, T obj) {
7         ...
8         if (iterable instanceof Bag<?>)
9             return ((Bag<E>) iterable).getCount(obj);
10        ...
11        return size(filteredIterable(emptyIfNull(iterable),
12            EqualPredicate.<E>equalPredicate(obj)));
13    }
14    ...
15 }

```

(b) Function that fetches frequency of an object from an Iterable object from apache-commons

```

1 class ComparatorUtils {
2     ...
3     static <E> E min(E o1, E o2, Comparator<E> comparator) {
4         int c = comparator.compare(o1, o2);
5         return c < 0 ? o1 : o2;
6     }
7     ...
8 }

```

(c) Method that returns minimum of two objects based on a Comparator from apache-commons

Figure 6.3 Functions from Open-Source Java library from commons by apache and guava by google and Python library collections

identified as the most similar code snippet. Fig. 6.3b shows a function that checks for the number of occurrences of an object in an Iterable object. Although it would seem that Fig. 6.3a and Fig. 6.3b are behaviourally similar, in practice they are not since a Set is not an Iterable object.

With respect to AST, Fig. 6.3c from Apache commons-lang is very similar to Fig. 6.3a. Although extremely similar based on their ASTs, they are functionally different. Fig. 6.3c uses a Comparator to identify the minimum of two numbers while Fig. 6.3a returns the frequency of an element in a MultiSet.

If both AST and token based similarities are used in COSAL, Fig. 6.3d is the most similar to Fig. 6.3a and share similar behaviour as well. Fig. 6.3d is method from the AbstractMapMultiSet class from

```

1 class AbstractMapMultiSet<E> {
2     ...
3     /**
4     * ... frequency ...
5     */
6     public int getCount(final Object object) {
7         final MutableInteger count = map.get(object);
8         if (count != null)
9             return count.value;
10        return 0;
11    }
12    ...
13 }

```

(d) Method that returns count of a AbstractMapMultiSet from apache-commons

```

1 class Counter(dict):
2     """
3     ... count ...
4     """
5     ...
6     def __getitem__(key):
7         return self.get(key, 0)
8     ...

```

(e) Function to get count of a key from a Counter from collections library.

Figure 6.3 Functions from Open-Source Java library from commons by apache and guava by google and Python library collections (cont.)

Apache commons-collection which returns the count of an element from an AbstractMapMultiSet object. The AbstractMapMultiSet is an implementation of a MultiSet in apache commons-collection.

COSAL also identifies a similar code snippet to Fig. 6.3a from the collections library in Python. Fig. 6.3e returns the count of a an element from a Counter. A Counter is Pythoncollection which like a bag takes elements and maintains a count of the number of occurrences of each element. For this pair, we can see that they share few common tokens (count, get), do not have similar ASTs but are behaviorally similar. Hence, the token based and IO based similarity in COSAL influence the ranking of search results and returns Fig. 6.3e as a valid search result for the query Fig. 6.3a.

In Table 4.7 we see a low value of $COSAL_{SLACC}$ since only around 9% of the files in BigCloneBench had executable code to support SLACC. But in the case of the open-source projects, around 68.2% of the Java and all the Python classes had executable code. The presence of dependent code in the libraries compared to the isolated files in BigCloneBench facilitated behavioural search in COSAL.

COSAL can be scaled to support other open source projects in the current implementation. The token based and AST based similarity measures for COSAL can be used on any project or file(s) in its current version. Since, the behavioural similarity measures used by COSAL is heavily dependent on SLACC, scaling to support new projects would require the projects have all its dependencies satisfied and executable.

Table 6.2 Performance of FaCoY and GitHub Search compared to COSAL based on 4,984 code snippets from BigCloneBench [94] benchmark which can be executed.

	Search	MRR	P@1/3/5/10	R@1/3/5/10
<i>SotP</i>	GitHub	68	64/58/54/46	64/68/72/75
<i>SotA</i>	FaCoY	79	74/70/68/57	74/76/81/84
<i>Single</i>	COSAL _{SLACC}	82	81 /78/74/67	81 /83/89/94
<i>Multi</i>	COSAL _{static}	80	78/75/72/66	79/83/87/91
<i>Sim.</i>	COSAL	83	81/79/74/68	81/86/91/96

6.2.2 On the Cost/Benefit of Dynamic Analysis

In section 4.9.3 and Table 4.7, we observe a low scores for code search using IO based similarity (COSAL_{SLACC}) compared to other techniques due to the small sample of files in BigCloneBench (9%) with executable code. To study the impact of semantic contribution of COSAL, we repeat the validation study on BigCloneBench but restricted to the files which can be executed (4,984). The results for this study are tabulated in Table 6.2.

In general, the results on the executable dataset are slightly better for all the techniques compared to the complete BigCloneBench dataset (Table 4.7). Although COSAL_{SLACC} is slightly better than COSAL_{static} compared to using static similarity based code search using AST and tokens (COSAL_{static}), it comes at a much larger cost. Executing snippets based on IO takes much larger time and memory eventually making code search slow and possibly impractical if the runtime data for the snippets are not cached. Since the gains are not very high with respect to the BigCloneBench dataset, it might be sufficient to rely on static similarity based code search in this case. Also, combining SLACC with COSAL_{static} does not improve the results of code search by a large margin. This further strengthens the argument to rely on static similarity measures for code search on BigCloneBench.

However, this cannot be generalized across datasets. BigCloneBench is built on Java code from open-source projects. When we search across programming languages (as seen in Table 4.5), using dynamic and static search similarities vastly improves the results for code search. This can be attributed to the syntactic differences between languages which can be overcome in many cases with semantic information [47]. Hence, the benefit of including dynamic similarity data must be balanced against the cost and context.

6.2.3 Support for new languages

COSAL supports Java and Python. While we have not demonstrated scalability to new languages, we comment on the effort required.

For dynamic behavior, COSAL is dependent on SLACC [134], so adding a new language to the former requires support in the latter. However, COSAL_{static} can be extended to new languages by

adapting the token and AST analyses.

A language-specific tokenizer like `c-tokenizer` [146] or a generic grammar based tokenizer like ANTLR [76] can be used to read source code and convert it into tokens using the approach detailed in from section 4.2.1. For the AST, COSAL uses a generic AST to represent source code across different languages. Using a language-specific AST Parser like `clang` for C [148] or `roslyn` for .NET [143], code should be parsed and converted to the generic AST based on the grammar available in the GitHub code repository for COSAL [135]. If a feature specific to a language is not supported by the grammar a new node can be created based on the feature's syntactic structure.

6.2.4 Adding new search similarities

COSAL uses three search similarity objectives for multi-objective search. New search objectives can be added or existing search objectives can be replaced in COSAL. First, a similarity measure to compare code snippets has to be defined. The similarity measure has to be a numerical value as the search results are ordered using it. Next, an index must be created characterizing the code in the repository. Once the similarity measure and the search index is defined, the COSAL configuration file has to be updated to enable the search similarity measure.

6.3 Embedding code similarity

We have evaluated StaCE against prior work in embedding based and index based code-to-code search and clone detection with techniques. Both the types of techniques have their strengths and limitations and are more effective in specific situations. In this section, we analyse the results of StaCE, the impact of dynamic similarity and a hybrid approach to address their limitations and threats to the validity.

6.3.1 Analyzing Results

In section 4.9 we see that StaCE can be used for scalable code-to-code search. This section presents an example from the AtCoder dataset illustrating cross-language code-to-code search. Fig. 6.4 contains snippets from four submissions for the *Connectivity* problem from the AtCoder dataset. In this problem, the participants are asked to find the number of nodes connected to each other in bi-directional graph network. A Disjoint Set (also known as UnionFind) data structure is commonly used to find the connectivity in graphs. 6.4a contains the `find` and `union` functions. The `find` function is recursive and the `union` function sets the parent of the second node as the common parent between the two nodes. 6.4c is a java submission that uses the same logic as 6.4a but have different variable and function names. The `lookup` and `unite` functions from 6.4c is similar to the `find` and `union` functions from 6.4a respectively. In 6.4d, the data-structure is called `DisjointSet` and the `find` function has the same implementation as the `parent` function from 6.4c. However, the `union` function uses an auxiliary `size` variable to ensure that the data-structure is balanced. This makes it more optimized compared to the `unite` and `union` functions from 6.4c and

```

6 def find(parent, x):
7     if parent[x] == x:
8         return x
9     else:
10        parent[x] = find(parent, parent[x])
11        return parent[x]
12
13 def union(parent, x, y):
14     find(parent, x)
15     find(parent, y)
16     if parent[x] != parent[y]:
17         parent[parent[y]] =
            ↪ parent[parent[x]]

```

(a) Python: Union Find from query

```

28 class UF {
...
31     int find(int x) {
32         int p = x;
33         while (id[p] != p)
34             p = id[p];
35         return p;
36     }
37     int union(int x, int y) {
38         x = find(x); y = find(y);
39         if (x != y)
40             id[x] = y;
41         return y;
42     }
43 }

```

(b) Java: Union Find with union by rank

```

227 class UnionFindTree {
...
244     int lookup(int x) {
245         if (par[x] == x) { return x;}
246         else {return par[x] =
            ↪ lookup(par[x]);}
247     }
248     void unite(int x, int y) {
249         x = lookup(x); y = lookup(y);
250         if (x == y) return;
251         par[y] = x
252     }
253 }

```

(c) Java: Union Find from query

```

11 class DisjointSet {
...
22     int find(int a) { ... }
...
28     void union(int a, int b) {
29         int pa = find(a); int pb =
            ↪ find(b);
30         if (pa == pb) return;
31         if (this.size[pa] >
            ↪ this.size[pb]) {
32             this.size[pa] += this.size[pb];
33             this.parent[pb] = pa;
34         } else {
35             this.size[pb] += this.size[pa];
36             this.parent[pa] = pb;
37         }
38     }
39 }

```

(d) Java: Union Find with union by rank

Figure 6.4 Snippets from four valid solutions for the connectivity problem from AtCoder. All four snippets show implementations the Union-Find data structure.

6.4a respectively. In the last snippet 6.4b, the data-structure is called UF and the find function is implemented in an iterative manner. Hence, this find function is structurally different compared to the other three snippets.

When StaCE is used to lookup similar Java snippets to 6.4a, the search results contain 6.4c, 6.4d and 6.4b at ranks 1,4 and 6 respectively. 6.4a and 6.4c contain structurally identical components while the variable names have similar context and hence similar embeddings. COSAL_{static} on the other hand does not return 6.4c as the most similar snippet since 6.4a and 6.4c do not share any common tokens. 6.4d and 6.4a have similar find functions but slightly different structural similarities due to rank optimization of the union data-structure (Lines 31-37 in 6.4d). Finally, the structural difference between the two implementations of the find functions in 6.4a (recursive) and 6.4b (iterative) can be attributed for lower rank for 6.4b. In such cases, approaches using dynamic

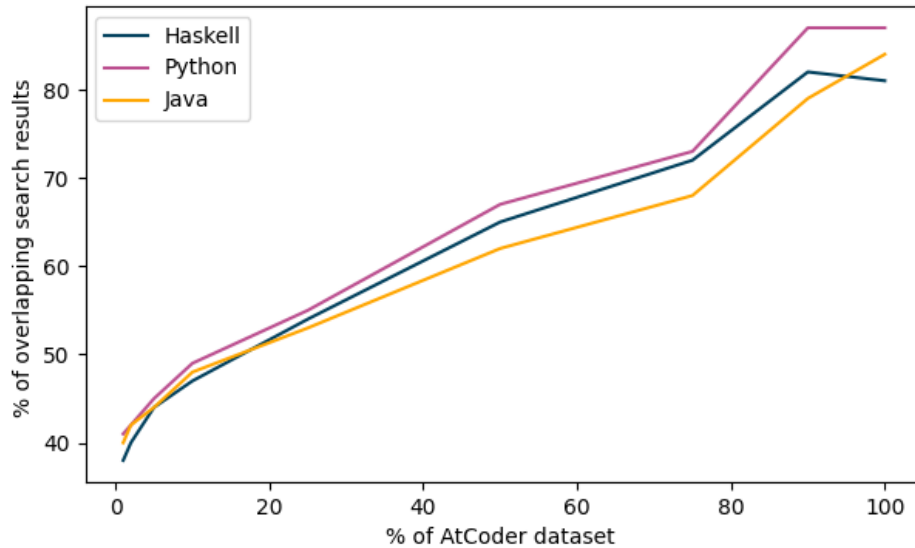


Figure 6.5 Percentage of overlapping results for StaCE vs COSAL for {1,2,5,10,20,50,100}% of data for 100 random queries in Haskell, Java and Python.

similarity measures tend to rank 6.4b higher. For example, if COSAL was used to lookup similar Java snippets to 6.4a, 6.4b is the third most similar search result.

6.3.2 Explaining Similarity

In sections section 4.9 and section 5.4.4, we see that both COSAL and StaCE have their merits and demerits. Both approaches have similar effectiveness in cross-language code-to-code search and code clone detection. StaCE is more scalable with faster runtimes and tolerable model building time. However, StaCE requires a large training data to achieve comparable results to COSAL and would be ineffective for smaller code repositories. On the other hand, COSAL has no training time, tolerable query time for smaller datasets and large query time for larger datasets. COSAL also provides the added advantage of explainability due to the choice of similarity measures: contextual similarities can be inferred by examining the common tokens; structural similarity can be inferred from visual difference in the ASTs; and behavioural differences can be highlighted by describing the common inputs and outputs. StaCE like other machine learning models yields results that lack explainability, are less interpretable and do not offer much transparency into process of arriving at the search results. Modern machine learning algorithms can be augmented with domain knowledge to overcome some of these limitations but it comes with additional time overhead and bookkeeping [136].

COSAL and StaCE can be used to overcome the limitations of each other. Fig. 6.5 shows the median percentage of overlapping search results for 100 random queries between COSAL and StaCE for varying data sizes. We use 1%, 2%, 5%, 10%, 25%, 50%, 75%, 90% and 100% of the AtCoder data for both the approaches. Queries are randomly selected from the test set across Haskell, Java

and Python. We observe that for large data sizes, there is a greater overlap between the search results across the three languages. Hence, a hybrid approach can be used for larger datasets to use the explainability of COSAL on StaCE. StaCE can be used to perform a high level search to filter out a subset of similar snippets to a given query. Subsequently, COSAL can be applied on the smaller subset to generate explainable differences between the source code. This is solution to address scalability of COSAL and the explainability of StaCE. We hope to explore such a hybrid approach in the future.

6.3.3 Dichotomy of Dynamic Similarity

COSAL uses three similarity measures to compare code snippets: d_{token} for context, d_{AST} for structure, and d_{IO} for behavior. We observe that using COSAL with IO-based similarity ($COSAL_{SLACC}$) yields higher precision but lower recall compared to other similarity metrics ($COSAL_{token}$ and $COSAL_{AST}$), particularly for Java and Python queries. We also note that augmenting static similarity measures with d_{IO} greatly improves the precision of COSAL.

In contrast, StaCE uses only tokens and AST based similarity measures and does not rely on dynamic similarity to generate code vectors with comparable (and sometimes better) results to COSAL. A natural intuition would be to vectorize snippets of code with IO similarity as well. However, the representation of behaviour using IO and the large input space due to random fuzzing of SLACC. To overcome this limitation, in StaCE, we concatenate the IO similarity between two snippets as computed by COSAL with the tree and tokens vector. The results for this approach are also highlighted in Table 4.5 as StaCE+IO. The results using this approach is similar to StaCE for Java and slightly worse for Haskell and Python. This shows that using this particular representation of IO based similarity is not effective when vectorizing code.

Executability is required to compute dynamic similarity. In this study, we observed 54% of the Haskell snippets, 69% of Java and 82% of the Python snippets from AtCoder dataset was executable. In contrast our prior study on BigCloneBench [94] dataset found that only 9% of the Java snippets were executable since we could not resolve the source code dependencies. Since the gains with the inclusion of IO based similarity are not very high and due to the reliance on runtime data, it might not be effective for large scale datasets. However, if users prefer the inference of varying inputs, we recommend COSAL which offers better explainability. In future, we will look at exploring alternate representations of behavioral similarity that do not rely on IO relationships [22, 101].

CHAPTER

7

RELATED WORK

7.1 Code Similarity

Source code similarity is used to characterize the relationship between pieces of code in software engineering applications such as program repair [53, 63, 74, 85, 89], code search [115, 123, 127], software security [40, 77, 119] and identifying plagiarized code [14]. In a survey by Roy et al. [49], code similarity techniques can broadly be categorized as static [18, 26, 30, 38, 43, 61], which represent structural similarities, and dynamic [47, 57, 101, 102], which represent behavioral similarities.

7.1.1 Static Similarity

Token-based approaches were one of the earliest techniques in identifying code clones [26, 30]. In such techniques tokens are extracted from the code snippets and converted into intermediate vector representations. Then the vector space distance between these vectors are used as a proxy for the similarity between the code clones. Although these methods were extremely fast, a major drawback in them are the large number of false positives. This can be attributed to the loss of context of code in the tokenization phase.

On the other hand, AST-based approaches [18, 38] use the Abstract Syntax Tree of the code snippet as the intermediate representation. These techniques captures the structural equivalence between clones [49] but it fails in cases where statements are reordered or in case of different representations of the same behaviour.

Finally, graph-based approaches [43, 61] use graphs constructed from the code snippets as intermediate representations. These graphs are generally constructed using the Abstract Syntax Tree

and subsequent Data Flow Analysis (DFA) of the code snippet. Then, graph matching algorithms are used to compare the graphs such that code clones are hypothesized to have greater match compared to arbitrary snippets of code. These approach encapsulates some semantic aspects as well since it uses DFA as part of the intermediate representation.

7.1.2 Dynamic Similarity

EQMiner [47] is the closest related work with respect to our methodology. They examined the Linux Kernel v2.6.24 by using a similar segmentation procedure, used 10 randomly generated inputs to execute them, and cluster based on IO behavior. Compared with SLACC, EQMiner crucially ignores cross-language code similarity. Furthermore, the implementation of EQMiner contains several limitations, noted by Deissenboeck [56], that make cross-language similarity infeasible and even replication itself impractical. As a result, we build on the ideas pioneered by EQMiner, while overcoming limitations in its original design. We introduce novel contributions, such as using grey-box analysis to overcome the limitations of simple random random testing, scale the input generation phase from 10 to 256 inputs, which drastically reduces false positives, introduce several steps and components to support complex language features, such as lambda functions, and handle differences arising from cross-language types. Finally, SLACC introduces flexibility in clustering as it permits a tolerance on similarity due to the SIM_T hyper-parameter.

HitoshiIO [102] by Su et al. also performs simion-based comparisons to identify similar code. It uses existing workloads like test-cases or ‘main’ function calls to collect values for the behavior rather than the random testing approach proposed in EQMiner or the grey-box analysis approach used in SLACC. Research shows that existing unit tests do not attain complete code coverage [81] and as a result, the application of such a technique to open source repositories might not be produce a comprehensive set of similar code snippets. This conjecture can be observed in RQ1 where SLACC identifies more clones to HitoshiIO by an order of magnitude. Further, HitoshiIO operates at a method level granularity while SLACC can operate at method or statement level granularity. Naturally, this ensures a greater number of code clones since SLACC can identify succinct behavior in complex code snippets.

7.2 Code-to-code search

In code search, the goal is to find code that is similar to a given query. Historically, developers have preferred general search engines such as *Google* and *Bing* when searching for code to reuse [55, 85, 100]. Some code search tools [70, 153] use code snippets as the query, a problem called code-to-code search. Solutions to code-to-code search vary in several dimensions. In this work, we focus on three: within [114, 115] vs. between languages [123–125], static [26, 38, 153] vs. dynamic analysis [48, 134], and index-based [115, 123, 156] vs. model based [114, 124, 125] approaches.

In cross-language code-to-code search, the query is a code snippet in one source language and the results are from a different target language(s). AROMA [123], supports cross-language code-to-

code search across Java, Hack, JavaScript, and Python using static analysis based on the parse tree. Since AROMA is not publicly available, it is not used as a baseline in this study. InferCode [138] is a self supervised cross-language (Java, C, C++ and C#) code representation approach using Tree-based Convolutional Neural Networks based on syntax subtrees.

For a single language, static analysis, and index-based approach, perhaps the most successful behavioral code-to-code search technique is FaCoY [115], which can find more Type-4 clones in the BigCloneBench [94] benchmark than prior approaches. Amongst cross-language, static analysis, and model-based approaches, Perez and Chiba [125] use a semi-supervised learning approach to create a model to embed code snippets on a vector space and create a mapping between Java and Python.

7.2.1 Embedding Source Code

DeepCS [114] is a Deep Neural Network model based code-to-code search tool using a custom CODenn architecture for Java code. The tool jointly embeds method body and documentation (javadoc) for each method in a code corpus into a high dimensional vector space. Once the model is built, a query is searched by first projecting onto the vector space and then returns snippets that have the top K nearest vectors to the query vector. Since DeepCS depends heavily on documentation of code and specific to Java, it is not applicable for cross-language code-to-code search.

InferCode [138] is a self supervised cross-language (Java, C, C++ and C#) code representation approach using Tree-based Convolutional Neural Networks based on syntax subtrees. InferCode supports languages that are limited supported by SrcML [67] and the current version is not extendable to Python and Haskell. Hence we do not benchmark COSAL and StaCE against InferCode. We plan to return to this future studies.

There are many (and sophisticated) neural network designs that can be used as a replacement for our vectorizing approach, such as ASTNN [131], GGNN [105] or CodeBERT [132]. However, these approaches are not generalizable for different programming languages. ASTNN splits large ASTs from C and Java into smaller sequence trees and encodes them into vectors using a bidirectional RNN. GGNN uses graph based representation and neural networks to vectorize code. However, GGNN is limited to C#, requires additional engineering effort and larger hardware to support more complex and diverse graphs. CodeBERT builds a bimodal representation for Natural Language and Programming Language using a multi layer Transformer architecture. It contains representations for Java and Python but cannot be used for cross-language code to code search or cross-language code-clone detection.

A small number of tools support cross-language code clone detection using code vectors [108, 124, 125]. API2Vec [108] detects clones between two syntactically similar languages by embedding source code into a vectors and subsequently comparing the similarity between the vectors. CLCDSA [124] identifies nine features from the source code AST and uses a deep neural network to learn the features and detect cross language clones. Perez and Chiba [125] propose an LSTM-based deep learning architecture using ASTs to detect clones in Java and Python code. These three tools

build machine learning models to detect code clones. As a result, these techniques require a large number of annotated training data to build the model and the hyper-parameters need to be carefully optimized to avoid over-fitting.

7.3 Clone Detection

Clone detection is a special case of code-to-code search; results are identified as clones if they meet a specified similarity threshold. Clones are often categorized into four types: types I-III are based on syntax and type IV is based on behavior.

Most code clone detection tools [18, 26, 30, 38, 43, 47, 61, 102] have been proposed for single language clone detection and on static typed languages like Java [35, 38] and C [11, 18, 26, 38]. A small number of tools support cross-language code clone detection [108, 124, 125, 134]. API2Vec [108] detects clones between two syntactically similar languages by embedding source code into a vectors and subsequently comparing the similarity between the vectors. CLCDSA [124] identifies nine features from the source code AST and uses a deep neural network to learn the features and detect cross language clones. Perez and Chiba [125] propose an LSTM-based deep learning architecture using ASTs to detect clones in Java and Python code. These three tools build machine learning models to detect code clones. As a result, these techniques require a large number of annotated training data to build the model and the hyper-parameters need to be carefully optimized to avoid over-fitting.

CHAPTER

8

FUTURE WORK

Similarity of code can be used in multiple other applications which are beyond the scope of this thesis proposal. This chapter highlights some of these avenues of expanding source code similarity and its applications. It represents ideas that are outside the scope of the thesis but are nevertheless interesting.

8.1 Learning Programming Languages using Examples

A study by Mayer and others in 2017 found that an engineer who makes an open source contribution knows 7 Programming languages on an average and knowledge of one programming language can help learn a new language faster [107]. Programmers prefer using cross-language learning strategy based on analogous features as the most common approach when learning a new programming language [117]. They also prefer examples of code over official documentation when learning new languages [95]. We ask,

RQ 13: *Can examples based on dynamic similarity help programmers in learning a new language compared to examples based on static similarity?*

Middleton and Stolee in their empirical study on code clone analysis find that 18 out of the 49 participants suggested that more dynamic information would help them better identify cross-language code clones [150]. We use this study as a motivation to compare examples based on static and dynamic similarity in learning new programming languages.

RQ 14: *Does augmenting static similarity with dynamic similarity in recommending examples help programmers compared to independently using static or dynamic similarity?*

In our study in multi-objective code search we observed that augmenting dynamic similarity based code search with static similarity results in better search results section 4.9.1. In this study, we augment static information like complexity of code, maintainability index and coupling to dynamic code examples. We then examine if such examples help programmers compared to static or dynamic information.

RQ 15: *What features in code examples do developers prefer while learning a new programming language?*

In this research question we aim on identifying features in the examples recommended to developers while learning a new programming language. Some of the features can include

- **Verbosity:** Since dynamic examples are recommended based on Input output behaviour, similar examples can be extremely detailed and verbose or can call an external API to perform the same task.
- **Code complexity:** Similarly, code examples can have different time complexities with a trade-off with comprehension. For example, consider a pair of code example to sort an array of integers. Examples can include BubbleSort which is easy to read but has a time complexity of $O(n^2)$ or TimSort which has a time complexity of $O(n \log(n))$ but is complicated since it combines binary insertion sort and merge sort which might make it hard first step when learning a new language.

8.2 Performance based Refactoring

While profiling IO behaviour of code snippets in SLACC, we also profile the snippets for runtime. This information along with memory profiling of the snippets can be used to refactor less efficient code [29, 137]. We ask,

RQ16: *Performance Study*

In this study, we measure the percentage of refactored code with respect to its validity. We will also measure the performance changes of the refactored code in terms of runtime and memory consumed.

RQ17: *Usability Study*

To evaluate usability the refactoring tool, we will conduct a controlled experiment with participants. We will test the hypothesis: For a sub-optimal code snippet, the

dynamic similarity based refactoring tool can the participants find better optimization possibilities to obtain a better performing version of the code snippet.

8.3 Automated Test Suite Generation

In section 3.2 we use dynamic analysis to find similarity of code snippets. As a consequence, this analysis identifies critical inputs where behaviour of the snippets start changing. These critical inputs and values in its domain, can be used to generate test cases for a functions as they represent the bellwether behaviour of such code snippets. Subsequently, this information can be used to improve the test suite for the code snippets in question [109]. In this study, we use dynamic analysis of source code code to automatically generate test suite. We evaluate,

RQ18: Quality of Test Suite

We will be measuring the quality of the test suite using statement coverage [17], branch coverage [17] and mutation analysis [50].

- **Statement Coverage** measures the percentage of the executed statements to the total number of statements in the application under test.
- **Branch Coverage** measures the percentage of executed conditional branch statements to the total number of statements in the application under test.
- In **Mutation Analysis** or Mutation Testing, a syntactic change (mutation) is made to the source code under test and executed against the test-bench. If the mutation is not detected by the test-bench, then any behaviour altered by the mutation has not been sufficiently exercised.

RQ19: Usability of Test Suite

We will assess the usability of the test suite by empirically comparing it to standard test suite generators EvoSuite [52] and AFL [121].

8.4 Cross-language code search using distributed code representations

In Chapter 3 code was represented based on their IO behaviour while and [108, 120, 125]. In this study, we propose on embedding tokens and ASTs as vectors on a common search space and subsequently using it for code-to-code search.

RQ20: Can tokens and AST for code be embedded onto the same vector space?

Common embedding space has been used in applications like labeling images [103]. In software engineering, it has been used in applications like generating documentation

for source code [111, 120]. In this study we develop a common embedding space for tokens and AST using an LSTM deep learning model.

RQ21: Can a common vector representation for tokens and AST be used in code-to-code search within a language?

Using the token and AST embedding, like section 4.9.3, we study if it can be used for code-to-code search in a programming language.

RQ22: Can a common vector representation for tokens and AST be used in cross-language code-to-code search?

Using the model developed as part of the previous research question, like section 4.9.2, we study if it can be used for cross-language code-to-code.

8.5 Cross-language code similarity for open-source code

SLACC, COSAL and StaCE were primarily not evaluated on open-source code in a cross-language context. Like other contemporary tools ASTLearner [125], CLCDSA [124] and HitoshiIO [102], we rely on submissions from programming contests to evaluate the tools proposed in this thesis across programming languages. These code snippets contain a wide range of features shared between languages (like loops and conditional statements) and unique to languages (`list-comprehension` in Python). However, they are not representative of real world code [138].

To demonstrate the merits of tools for cross-language similarity, we require snippets from open-source projects. BigCloneBench [94] is one of the largest publicly available code clone benchmarks for Java. It contains over 55,000 source code files harnessed from approximately 25,000 open source repositories and grouped into 43 distinct functionalities. This dataset is used as a benchmark for code search [115] and clone detection [99] techniques. A similar approach could be used to curate a benchmark of snippets across languages, but at the time of this research, such a benchmark was not available.

Recently, Bui et al. [138] use code from Rosetta¹ and Github to develop a small repository of similar code from Java, C, C++ and C#. The authors use matching keywords to identify similar snippets. Although this is a good start in our experiments we realized that this approach is not applicable for languages that follow different programming paradigms. Matching keywords as a measure of similarity led to low recall while constructing a similar dataset for Java, Python and Haskell. However, this is a good start and provides avenues for building a more representative open-source dataset from similar code across languages.

¹<http://www.rosettacode.org>

8.6 TransCompiler

Transcompilers or source-to-source compiler, is a tool that converts code from one programming language to another programming language on a similar level of abstraction. This is useful when porting code between different platforms like Android to IOS. This is also common during acquisitions when repositories are migrated from one language to another [116]. In this study, we look into migration of code from one high level language to another using dynamic similarity of code.

RQ23: Can dynamic similarity of code be used in translating code from a high level language to another high level language?

TransCoder, the state-of-the-art transpiler uses unsupervised learning based on static information to translate code between Java and Python [133]. Due to this reason, the code fails to translate similar idioms like `list-comprehension` in Python and `for` loop in Java. Using dynamic similarity (section 4.2.3) and AST based similarity (section 4.2.2), such equivalences can be inferred as shown later in section 4.9.2. Hence, in this study we will explore if dynamic similarity augmented with static similarity can be used for better translation of code.

CHAPTER

9

CONCLUSION

This work presents research on similarity of source code based on syntactic and semantic measures. Similarity based metrics are used across multiple software engineering applications in software engineering. This work describes two such applications of code clone detection and code search and proposes how source code similarity can be applied in learning new programming languages.

We first present SLACC which precisely yields semantic code clones across programming languages. SLACC is the first work to identify semantic code clones in a dynamic typed language and across programming languages. SLACC identifies clones by comparing the IO relationship of segmented snippets of code from a target repository. Input values for the segmented code are generated using multi-modal grey-box fuzzing. This results in fewer false positives compared to current state of the art semantic code clone detection tool, HitoshiIO. The generated inputs are also memoized, which makes SLACC scalable. In our study, we identify code clones between Java and Python from Google Code Jam submissions. Compared to HitoshiIO, SLACC identifies significantly (6x) more code clones, with greater precision (86.7% vs. 30.7%). Furthermore, SLACC identifies partial fine-grain code clones with succinct behavior in the target repository as SLACC segments the target repository using pre-order traversal and a recursive sliding window that accounts for nested code blocks and inline statements such as lambda functions. This observation is verified in our study, as most of the code clones have two arguments or less. Further, more than half the code clones have less than 6 lines and 5 lines for Java and Python respectively. SLACC detects code clones in a multi-language code corpora using some restrictions on the data-types (and their bounds) between the languages. That said, the number of clones detected was fewer and the number of false positives were slightly more compared to code clones within the same language. But, by broadening the support for more features, these metrics can be improved.

Next we present COSAL, a cross-language code-to-code search tool that uses static and dynamic analyses without a machine learning model. For static analysis, we used two similarity measures based on extracted tokens from source code and a tree edit distance based on a generic AST. For dynamic analysis, we used SLACC to compute IO similarity between segments of code. For a given code search query, these three similarity measures find similar code using multi-objective search. Our experimental evaluation on Java, Python and Haskell files from AtCoder shows that COSAL outperforms state-of-the-art code search tools FaCoY and Code2Vec and industrial benchmark of GitHub code search with respect to *precision@{1,3,5,10}* and *recall@{1,3,5,10}*. We also compare COSAL to state-of-the-art clone detection techniques using the same AtCoder dataset and find that COSAL has better *recall* and *F1 score*.

Finally, we present StaCE, a code embedding technique using features based on context and structure. We extend the token based representation and language agnostic structural representations of COSAL to generate code embeddings. Using code snippets across Java, Python and Haskell from the AtCoder dataset, we show that StaCE outperforms state-of-the-art cross-language code search tools Code2Vec and industrial benchmark of GitHub code search. We also compare StaCE to state-of-the-art static and dynamic code clone detection techniques find that StaCE is better than static clone detection techniques and has better Recall and F-measure to dynamic clone detection techniques. Finally, we compare StaCE to COSAL, an index based approach on varying data sizes and find that StaCE has smaller query times compared to COSAL on larger datasets. Embedding multiple representations of source code helps overcome the need for slower similarity measures like dynamic similarity that requires executable code and sorting across multiple similarity measures.

Applications to cross-language code similarity extend beyond code-to-code search and code clone detection. We suggest how this work on cross-language code similarity can be extended to applications of learning new programming languages and code refactoring. Hence, cross-language source code similarity appears to have a bright future, but more work is needed to evaluate it for more languages and in relevant applications, such as language translation, language migration, and program repair.

References

- [1] Luntz, A. "On estimation of characters obtained in statistical procedure of recognition". *Technicheskaya Kibernetika* **3** (1969).
- [2] Bentley, J. L. "Multidimensional binary search trees used for associative searching". *Communications of the ACM* **18.9** (1975), pp. 509–517.
- [3] Cousot, P. & Cousot, R. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints". *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM. 1977, pp. 238–252.
- [4] Bowles, S. W. & Bethke Jr, G. E. *Multi-pass system and method for source to source code translation*. US Patent 4,374,408. 1983.
- [5] Kirkpatrick, S. et al. "Optimization by simulated annealing". *science* **220.4598** (1983), pp. 671–680.
- [6] Michalski, R. S. "A theory and methodology of inductive learning". *Machine learning*. Springer, 1983, pp. 83–134.
- [7] Ferrante, J. et al. "The program dependence graph and its use in optimization". *ACM Transactions on Programming Languages and Systems (TOPLAS)* **9.3** (1987), pp. 319–349.
- [8] Zhang, K. & Shasha, D. "Simple fast algorithms for the editing distance between trees and related problems". *SIAM journal on computing* **18.6** (1989), pp. 1245–1262.
- [9] Scholtz, J. & Wiedenbeck, S. "Learning second and subsequent programming languages: A problem of transfer". *International Journal of Human-Computer Interaction* **2.1** (1990), pp. 51–72.
- [10] Wu, Q. & Anderson, J. R. *Problem-solving transfer among programming languages*. Tech. rep. Carnegie Mellon University, 1990.
- [11] Yang, W. "Identifying syntactic differences between two programs". *Software: Practice and Experience* **21.7** (1991), pp. 739–755.
- [12] Baldi, P. & Chauvin, Y. "Neural networks for fingerprint recognition". *Neural Computation* **5.3** (1993), pp. 402–418.
- [13] Fonseca, C. M., Fleming, P. J., et al. "Genetic Algorithms for Multiobjective Optimization: Formulation Discussion and Generalization." *Icga*. Vol. 93. July. Citeseer. 1993, pp. 416–423.
- [14] Baker, B. S. "On finding duplication and near-duplication in large software systems". *Proceedings of 2nd Working Conference on Reverse Engineering*. IEEE. 1995, pp. 86–95.
- [15] Kennedy, J. & Eberhart, R. "Particle swarm optimization". *Proceedings of ICNN'95-International Conference on Neural Networks*. Vol. 4. IEEE. 1995, pp. 1942–1948.
- [16] Parr, T. J. & Quong, R. W. "ANTLR: A predicated-LL (k) parser generator". *Software: Practice and Experience* **25.7** (1995), pp. 789–810.

- [17] Zhu, H. et al. "Software unit test coverage and adequacy". *ACM computing surveys (csur)* **29.4** (1997), pp. 366–427.
- [18] Baxter, I. D. et al. "Clone detection using abstract syntax trees". *Software Maintenance, 1998. Proceedings., International Conference on.* IEEE. 1998, pp. 368–377.
- [19] Deng, K. "Omega: On-line memory-based general purpose system classifier". PhD thesis. Carnegie Mellon University, 1998.
- [20] Patenaude, J.-F. et al. "Extending software quality assessment techniques to java systems". *Proceedings Seventh International Workshop on Program Comprehension.* IEEE. 1999, pp. 49–56.
- [21] Komondoor, R. & Horwitz, S. "Using slicing to identify duplication in source code". *International static analysis symposium.* Springer. 2001, pp. 40–56.
- [22] Krinke, J. "Identifying similar code with program dependence graphs". *Proceedings Eighth Working Conference on Reverse Engineering.* IEEE. 2001, pp. 301–309.
- [23] Beit-Aharon, J. *Source code translation.* US Patent App. 15/894,096. 2002.
- [24] Burd, E. & Bailey, J. "Evaluating clone detection tools for use during preventative maintenance". *Proceedings. Second IEEE International Workshop on Source Code Analysis and Manipulation.* IEEE. 2002, pp. 36–43.
- [25] Deb, K. et al. "A fast and elitist multiobjective genetic algorithm: NSGA-II". *IEEE transactions on evolutionary computation* **6.2** (2002), pp. 182–197.
- [26] Kamiya, T. et al. "CCFinder: a multilinguistic token-based code clone detection system for large scale source code". *IEEE Transactions on Software Engineering* **28.7** (2002), pp. 654–670.
- [27] Greenspan, M. & Yurick, M. "Approximate kd tree search for efficient ICP". *Fourth International Conference on 3-D Digital Imaging and Modeling, 2003. 3DIM 2003. Proceedings.* IEEE. 2003, pp. 442–448.
- [28] Gupta, D. "What is a good first programming language?" *Crossroads* **10.4** (2004), pp. 7–7.
- [29] Higo, Y. et al. "Refactoring support based on code clone analysis". *International Conference on Product Focused Software Process Improvement.* Springer. 2004, pp. 220–233.
- [30] Li, Z. et al. "CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code." *OSdi.* Vol. 4. 2004, pp. 289–302.
- [31] Relf, P. "Achieving software quality through identifier names". *Qualcon 2004.* 2004, pp. 33–34.
- [32] Graves, A. & Schmidhuber, J. "Framewise phoneme classification with bidirectional LSTM and other neural network architectures". *Neural networks* **18.5-6** (2005), pp. 602–610.
- [33] Lahtinen, E. et al. "A study of the difficulties of novice programmers". *ACM Sigcse Bulletin* **37.3** (2005), pp. 14–18.

- [34] Guthrie, D. et al. "A closer look at skip-gram modelling." *LREC*. Vol. 6. Citeseer. 2006, pp. 1222–1225.
- [35] Koschke, R. et al. "Clone detection using abstract syntax suffix trees". *2006 13th Working Conference on Reverse Engineering*. IEEE. 2006, pp. 253–262.
- [36] Liu, C. et al. "GPLAG: detection of software plagiarism by program dependence graph analysis". *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2006, pp. 872–881.
- [37] Bellon, S. et al. "Comparison and evaluation of clone detection tools". *IEEE Transactions on software engineering* **33.9** (2007), pp. 577–591.
- [38] Jiang, L. et al. "Deckard: Scalable and accurate tree-based detection of code clones". *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society. 2007, pp. 96–105.
- [39] Sun, J. et al. "Image vectorization using optimized gradient meshes". *ACM Transactions on Graphics (TOG)* **26.3** (2007), 11–es.
- [40] Walenstein, A. & Lakhotia, A. "The software similarity problem in malware analysis". *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2007.
- [41] Zhang, Q. & Li, H. "MOEA/D: A multiobjective evolutionary algorithm based on decomposition". *IEEE Transactions on evolutionary computation* **11.6** (2007), pp. 712–731.
- [42] Fjeldberg, H.-C. "Polyglot programming". PhD thesis. Master thesis, Norwegian University of Science and Technology, Trondheim/Norway, 2008.
- [43] Gabel, M. et al. "Scalable detection of semantic clones". *Proceedings of the 30th international conference on Software engineering*. ACM. 2008, pp. 321–330.
- [44] Kraft, N. A. et al. "Cross-language Clone Detection." *SEKE*. 2008, pp. 54–59.
- [45] Bird, S et al. "Accessing text corpora and lexical resources". *Natural Language Processing with Python* (2009).
- [46] Jensen, S. H. et al. "Type analysis for JavaScript". *International Static Analysis Symposium*. Springer. 2009, pp. 238–255.
- [47] Jiang, L. & Su, Z. "Automatic mining of functionally equivalent code fragments via random testing". *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM. 2009, pp. 81–92.
- [48] Reiss, S. P. "Semantics-based code search". *2009 IEEE 31st International Conference on Software Engineering*. IEEE. 2009, pp. 243–253.
- [49] Roy, C. K. et al. "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach". *Science of computer programming* **74.7** (2009), pp. 470–495.

- [50] Jia, Y. & Harman, M. “An analysis and survey of the development of mutation testing”. *IEEE transactions on software engineering* **37.5** (2010), pp. 649–678.
- [51] Sammut, C. & Webb, G. I. “Leave-one-out cross-validation”. *Encyclopedia of machine learning* (2010), pp. 600–601.
- [52] Fraser, G. & Arcuri, A. “EvoSuite: automatic test suite generation for object-oriented software”. *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM. 2011, pp. 416–419.
- [53] Gopinath, D. et al. “Specification-based program repair using SAT”. *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2011, pp. 173–188.
- [54] Kim, H. et al. “MeCC: memory comparison-based clone detector”. *Proceedings of the 33rd International Conference on Software Engineering*. ACM. 2011, pp. 301–310.
- [55] Sim, S. E. et al. “How well do search engines support code retrieval on the web?” *ACM Transactions on Software Engineering and Methodology (TOSEM)* **21.1** (2011), pp. 1–25.
- [56] Deissenboeck, F. et al. “Challenges of the dynamic detection of functionally similar code fragments”. *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*. IEEE. 2012, pp. 299–308.
- [57] Elva, R. & Leavens, G. T. *Jscracker: A semantic clone detection tool for java code*. Tech. rep. University of Central Florida, Dept. of EECS, CS division, 2012.
- [58] Elva, R. & Leavens, G. T. “Semantic clone detection using method ioe-behavior”. *2012 6th International Workshop on Software Clones (IWSC)*. IEEE. 2012, pp. 80–81.
- [59] Hindle, A. et al. “On the naturalness of software”. *2012 34th International Conference on Software Engineering (ICSE)*. IEEE. 2012, pp. 837–847.
- [60] Khan, M. E., Khan, F., et al. “A comparative study of white box, black box and grey box testing techniques”. *Int. J. Adv. Comput. Sci. Appl* **3.6** (2012).
- [61] Li, J. & Ernst, M. D. “CBCD: Cloned buggy code detector”. *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press. 2012, pp. 310–320.
- [62] Miettinen, K. *Nonlinear multiobjective optimization*. Vol. 12. Springer Science & Business Media, 2012.
- [63] Stolee, K. T. & Elbaum, S. “Toward semantic search via SMT solver”. *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM. 2012, p. 25.
- [64] Armstrong, T. G. et al. “LinkBench: a database benchmark based on the Facebook social graph”. *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM. 2013, pp. 1185–1196.

- [65] Arora, R. & Aggarwal, R. R. “Modeling and querying data in mongodb”. *International Journal of Scientific and Engineering Research* **4.7** (2013), pp. 141–144.
- [66] Binkley, D. et al. “The impact of identifier style on effort and comprehension”. *Empirical Software Engineering* **18.2** (2013), pp. 219–276.
- [67] Collard, M. L. et al. “srcml: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration”. *2013 IEEE International Conference on Software Maintenance*. IEEE. 2013, pp. 516–519.
- [68] Deb, K. & Jain, H. “An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part I: solving problems with box constraints”. *IEEE transactions on evolutionary computation* **18.4** (2013), pp. 577–601.
- [69] Graves, A. et al. “Speech recognition with deep recurrent neural networks”. *2013 IEEE international conference on acoustics, speech and signal processing*. IEEE. 2013, pp. 6645–6649.
- [70] Krugler, K. “Krugle code search architecture”. *Finding Source Code on the Web for Remix and Reuse*. Springer, 2013, pp. 103–120.
- [71] Krutz, D. E. & Shihab, E. “Cccd: Concolic code clone detection”. *Reverse Engineering (WCRE), 2013 20th Working Conference on*. IEEE. 2013, pp. 489–490.
- [72] Mikolov, T. et al. “Distributed representations of words and phrases and their compositionality”. *Advances in neural information processing systems*. 2013, pp. 3111–3119.
- [73] Mikolov, T. et al. “Efficient estimation of word representations in vector space”. *arXiv preprint arXiv:1301.3781* (2013).
- [74] Nguyen, H. D. T. et al. “Semfix: Program repair via semantic analysis”. *Software Engineering (ICSE), 2013 35th International Conference on*. IEEE. 2013, pp. 772–781.
- [75] Niwattanakul, S. et al. “Using of Jaccard coefficient for keywords similarity”. *Proceedings of the international multiconference of engineers and computer scientists*. Vol. 1. 6. 2013, pp. 380–384.
- [76] Parr, T. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.
- [77] Ray, B. et al. “Detecting and Characterizing Semantic Inconsistencies in Ported Code”. *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. ASE’13. Silicon Valley, CA, USA: IEEE Press, 2013, pp. 367–377.
- [78] Allamanis, M. et al. “Learning natural coding conventions”. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2014, pp. 281–293.
- [79] Bansal, G. & Tekchandani, R. “Selecting a set of appropriate metrics for detecting code clones”. *2014 Seventh International Conference on Contemporary Computing (IC3)*. IEEE. 2014, pp. 484–488.

- [80] Barr, E. T. et al. "The plastic surgery hypothesis". *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM. 2014, pp. 306–317.
- [81] Gopinath, R. et al. "Code coverage for suite evaluation by developers". *Proceedings of the 36th International Conference on Software Engineering*. ACM. 2014, pp. 72–82.
- [82] Milea, N. A. et al. "Scalable detection of missed cross-function refactorings". *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM. 2014, pp. 138–148.
- [83] Pennington, J. et al. "Glove: Global vectors for word representation". *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 2014, pp. 1532–1543.
- [84] Skudder, B. & Luxton-Reilly, A. "Worked examples in computer science". *Proceedings of the Sixteenth Australasian Computing Education Conference-Volume 148*. Australian Computer Society, Inc. 2014, pp. 59–64.
- [85] Stolee, K. T. et al. "Solving the search for source code". *ACM Transactions on Software Engineering and Methodology (TOSEM)* **23.3** (2014), p. 26.
- [86] Tomassetti, F. & Torchiano, M. "An Empirical Assessment of Polyglot-ism in GitHub". *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*. EASE '14. London, England, United Kingdom: ACM, 2014, 17:1–17:4.
- [87] Bruggen, D. van. *Javaparser - For processing Java code*. github.com/javaparser/javaparser. [Online; accessed 23-August-2019]. 2015.
- [88] Gormley, C. & Tong, Z. *Elasticsearch: the definitive guide: a distributed real-time search and analytics engine*. " O'Reilly Media, Inc.", 2015.
- [89] Ke, Y. et al. "Repairing programs with semantic code search (t)". *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE. 2015, pp. 295–306.
- [90] Mayer, P. & Bauer, A. "An Empirical Analysis of the Utilization of Multiple Programming Languages in Open Source Projects". *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*. EASE '15. Nanjing, China: ACM, 2015, 4:1–4:10.
- [91] Rush, A. M. et al. "A neural attention model for abstractive sentence summarization". *arXiv preprint arXiv:1509.00685* (2015).
- [92] Sadowski, C. et al. "How developers search for code: a case study". *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 2015, pp. 191–201.
- [93] Schroff, F. et al. "Facenet: A unified embedding for face recognition and clustering". *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 815–823.

- [94] Svajlenko, J. & Roy, C. K. “Evaluating clone detection tools with bigclonebench”. *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2015, pp. 131–140.
- [95] Acar, Y. et al. “You get where you’re looking for: The impact of information sources on code security”. *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2016, pp. 289–305.
- [96] Grover, A. & Leskovec, J. “node2vec: Scalable feature learning for networks”. *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*. 2016, pp. 855–864.
- [97] Melamud, O. et al. “context2vec: Learning generic context embedding with bidirectional lstm”. *Proceedings of the 20th SIGNLL conference on computational natural language learning*. 2016, pp. 51–61.
- [98] Mou, L. et al. “Convolutional neural networks over tree structures for programming language processing”. *Thirtieth AAAI Conference on Artificial Intelligence*. 2016.
- [99] Sajnani, H. et al. “Sourcerercc: Scaling code clone detection to big-code”. *Proceedings of the 38th International Conference on Software Engineering*. 2016, pp. 1157–1168.
- [100] Stolee, K. T. et al. “Code search with input/output queries: Generalizing, ranking, and assessment”. *Journal of Systems and Software* **116** (2016), pp. 35–48.
- [101] Su, F.-H. et al. “Code relatives: detecting similarly behaving software”. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM. 2016, pp. 702–714.
- [102] Su, F.-H. et al. “Identifying functionally similar code in complex codebases”. *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on*. IEEE. 2016, pp. 1–10.
- [103] Wang, L. et al. “Learning deep structure-preserving image-text embeddings”. *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 5005–5013.
- [104] White, M. et al. “Deep learning code fragments for code clone detection”. *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM. 2016, pp. 87–98.
- [105] Allamanis, M. et al. “Learning to represent programs with graphs”. *arXiv preprint arXiv:1711.00740* (2017).
- [106] Azzopardi, L. et al. “Lucene4IR: Developing information retrieval evaluation resources using Lucene”. *ACM SIGIR Forum*. Vol. 50. 2. ACM New York, NY, USA. 2017, pp. 58–75.
- [107] Mayer, P. et al. “On multi-language software development, cross-language links and accompanying tools: a survey of professional software developers”. *Journal of Software Engineering Research and Development* **5.1** (2017), p. 1.

- [108] Nguyen, T. D. et al. “Exploring API embedding for API usages and applications”. *Software Engineering (ICSE), 2017 IEEE/ACM 39th International Conference on*. IEEE. 2017, pp. 438–449.
- [109] Panichella, A. et al. “Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets”. *IEEE Transactions on Software Engineering* **44.2** (2017), pp. 122–158.
- [110] Xin, Q. & Reiss, S. P. “Leveraging syntax-related code for automated program repair”. *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2017, pp. 660–670.
- [111] Zheng, T. et al. “Auto-Documentation for Software Development”. *arXiv preprint arXiv:1701.08485* (2017).
- [112] Chung, Y.-A. & Glass, J. “Speech2vec: A sequence-to-sequence framework for learning word embeddings from speech”. *arXiv preprint arXiv:1803.08976* (2018).
- [113] Creswell, A. et al. “Generative adversarial networks: An overview”. *IEEE Signal Processing Magazine* **35.1** (2018), pp. 53–65.
- [114] Gu, X. et al. “Deep code search”. *Proceedings of the 40th International Conference on Software Engineering*. ACM. 2018, pp. 933–944.
- [115] Kim, K. et al. “FaCoY: a code-to-code search engine”. *Proceedings of the 40th International Conference on Software Engineering*. 2018, pp. 946–957.
- [116] Shepherd, D. C. “The cost-benefit analysis of usage data in robotstudio (keynote)”. *2018 IEEE Workshop on Mining and Analyzing Interaction Histories (MAINT)*. IEEE Computer Society. 2018, pp. 1–1.
- [117] Shrestha, N. et al. “It’s Like Python But: Towards Supporting Transfer of Programming Language Knowledge”. *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 2018, pp. 177–185.
- [118] Webber, J. & Robinson, I. *A programmatic introduction to neo4j*. Addison-Wesley Professional, 2018.
- [119] Yue, R. et al. “Automatic Clone Recommendation for Refactoring Based on the Present and the Past”. *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2018, pp. 115–126.
- [120] Alon, U. et al. “code2vec: Learning distributed representations of code”. *Proceedings of the ACM on Programming Languages* **3**.POPL (2019), pp. 1–29.
- [121] Goh, B. W. H. “American fuzzy lop (AFL) fuzzing” (2019).
- [122] Husain, H. et al. “Codesearchnet challenge: Evaluating the state of semantic code search”. *arXiv preprint arXiv:1909.09436* (2019).

- [123] Luan, S. et al. “Aroma: Code recommendation via structural code search”. *Proceedings of the ACM on Programming Languages* **3**.OOPSLA (2019), pp. 1–28.
- [124] Nafi, K. W. et al. “CLCDSA: cross language code clone detection using syntactical features and API documentation”. *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2019, pp. 1026–1037.
- [125] Perez, D. & Chiba, S. “Cross-language clone detection by learning over abstract syntax trees”. *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE. 2019, pp. 518–528.
- [126] Perry, D. M. et al. “SemCluster: clustering of imperative programming assignments based on quantitative semantic features”. *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2019, pp. 860–873.
- [127] Ragkhitwetsagul, C. & Krinke, J. “Siamese: scalable and incremental code clone search via multiple code representations”. *Empirical Software Engineering* **24.4** (2019), pp. 2236–2284.
- [128] Wikipedia contributors. *Levenshtein distance* — *Wikipedia, The Free Encyclopedia*. en.wikipedia.org/wiki/Levenshtein_distance [Online; accessed 23-August-2019]. 2019.
- [129] Wyrich, M. et al. “A theory on individual characteristics of successful coding challenge solvers”. *PeerJ Computer Science* **5** (2019), e173.
- [130] Xin, Q. & Reiss, S. P. “Revisiting ssFix for Better Program Repair”. *arXiv preprint arXiv:1903.04583* (2019).
- [131] Zhang, J. et al. “A novel neural source code representation based on abstract syntax tree”. *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE. 2019, pp. 783–794.
- [132] Feng, Z. et al. “Codebert: A pre-trained model for programming and natural languages”. *arXiv preprint arXiv:2002.08155* (2020).
- [133] Lachaux, M.-A. et al. “Unsupervised Translation of Programming Languages”. *arXiv preprint arXiv:2006.03511* (2020).
- [134] Mathew, G. et al. “SLACC: Simion-based Language Agnostic Code Clones”. *International Conference on Software Engineering (ICSE)* (2020).
- [135] *Multi Objective Code Search*. Zenodo, 2020.
- [136] Roscher, R. et al. “Explainable machine learning for scientific insights and discoveries”. *Ieee Access* **8** (2020), pp. 42200–42216.
- [137] Zhou, W. et al. “HARP: holistic analysis for refactoring Python-based analytics programs”. *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 2020, pp. 506–517.

- [138] Bui, N. D. et al. “InferCode: Self-Supervised Learning of Code Representations by Predicting Subtrees”. *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE. 2021, pp. 1186–1197.
- [139] Mathew, G. & Stolee, K. T. “Cross-Language Code Search Using Static and Dynamic Analyses”. ESEC/FSE 2021. Athens, Greece: Association for Computing Machinery, 2021.
- [140] Mathew, G. & Stolee, K. T. “Structural and Contextual Embeddings for Within-Language and Cross-Language Code Search”. *arXiv preprint arXiv:XXX* (2021).
- [141] AtCoder Inc. *AtCoder*. atcoder.jp. Accessed: 2020-08-12.
- [142] Community, P. *Python Keywords*. tiny.cc/q7jqsz. Accessed: 2020-08-12.
- [143] DotNet. *Roslyn*. github.com/dotnet/roslyn. Accessed: 2020-08-12.
- [144] Fau, A. & Bihler, R. *Java2CSharp*. sourceforge.net/projects/j2cstranslator/. Accessed: 2018-09-25.
- [145] Google. *Google Code Jam*. code.google.com/codejam. Accessed: 2018-09-25.
- [146] Halliday, J. *c-tokenzier*. github.com/substack/c-tokenizer. Accessed: 2020-08-12.
- [147] Krishnamurthi, S. & Fislser, K. “Programming paradigms and beyond” ().
- [148] Lattner, C. et al. *clang: a C language family frontend for LLVM*. clang.llvm.org. Accessed: 2020-08-12.
- [149] Mathew, G. et al. *SLACC*. github.com/DynamicCodeSearch/SLACC/tree/ICSE20. [Online; accessed 06-February-2020].
- [150] Middleton, J. & Stolee, K. T. *Confounds in Code Clone Comprehension*.
- [151] Oracle. *Java Language Keywords*. tiny.cc/s7jqsz. Accessed: 2020-08-12.
- [152] Python Community. *Python AST*. docs.python.org/3/library/ast.html. [Online; accessed 23-August-2019].
- [153] *SearchCode*. searchcode.com. [Online; accessed 06-February-2020].
- [154] Team GitHub. *GitHub Gist*. gist.github.com/discover. [Online; accessed 23-August-2019].
- [155] Team Github. *GitHub REST API*. docs.github.com/en/rest. Accessed: 2020-08-12.
- [156] Team Github. *GitHub Search*. tiny.cc/ig5nsz. Accessed: 2020-08-12.
- [157] Team Stack Overflow. *Stack Overflow*. stackoverflow.com. [Online; accessed 23-August-2019].
- [158] Wikipedia Contributors. *Java bytecode instruction listings*. en.wikipedia.org/wiki/Java_bytecode. [Online; accessed 23-August-2019].