Mapping Modern JVM Language Code to Analysis-friendly Graphs: A Pilot Study with Kotlin

Lu Li
School of Software Engineering
Tongji University
Shanghai, China
2033816@tongji.edu.cn

Yan Liu*
School of Software Engineering
Tongji University
Shanghai, China
yanliu.sse@tongji.edu.cn

***Abstract*—Kotlin is a modern JVM language, gaining adoption rapidly and becoming Android official programming language. With its widely usage, the need for code analysis of Kotlin is increasing. Exposing code semantics explicitly with a properly structured format is the first step in code analysis and the construction of such representation is the foundation for downstream tasks. Recently, graph-based approaches become a promising way for encoding source code semantics. However, current works mainly focus on representation learning with limited interpretability and shallow domain knowledge. The known evolvements of code semantics in new-generation programming languages have been overlooked. How to establish an effective mapping between naturally concise Kotlin source code with graph-based representation needs to be studied by analyzing known language features. In this paper, we propose a first-sight, rule-based mapping method, using composite representation with AST, CFG, DFG, and language features. We evaluate mapping strategies with ablation experiments by simulating a code search solution as a downstream task. Our graph-based method with built-in language features outperforms the text-based way without introducing greater complexity. By addressing the practical barriers to extracting and exposing the hidden semantics from Kotlin source code, our study also helps enlighten source code representations for other modern languages.**

Keywords- Kotlin; graph representation of code; code analysis; language feature;

I. INTRODUCTION

In 2017, Google announced Kotlin as an Android official programming language. It combines object-oriented and functional features. Being a more modern, expressive, and safer programming language, Kotlin has achieved a significant diffusion among developers, the adoption of the Kotlin was rapid. It was the fastest-growing language on GitHub 2018[1] and ranked 7th among mobile development languages in Top Programming Languages 2021 published by IEEE Spectrum[2].

As more and larger codebases written in Kotlin appear, the need for code analysis is increasing. Code analysis is learning from source code through relevant means to solve downstream tasks such as code defect detection, code search, etc., bringing lots of amazing tools and great convenience. The first step of machine learning-based approaches of code analysis is to map source code into intermediate code representation. Generated methods can be categorized as sequence-, tree-, and graph-based[3][4][5], among which graph-based representation has better performance in code analysis these years, becoming a more promising approach for intermediate representation.

However, there is a lack of code analysis related research based on Kotlin for now, and also no related research on the Kotlin graph representation method. Even the graph representation of code itself is still an issue under study, that there is no widely accepted method to convert code into graph representation[6]. Furthermore, Kotlin is a concise language, with less information presented in source code. The usage of language features of Kotlin is also under study. So, no one knows what is a more suitable way to map Kotlin source code to graph representation.

Therefore, in this paper, we conduct a pilot study on how to map Kotlin source code to analytics-friendly graphs. The contribution of this work is as follows:

- This is the first study focused on graph representation of Kotlin code to our knowledge. We proposed a first-sight rule-based feature-enhanced graph mapping method. It can provide other Kotlin downstream task researchers with a basis for constructing graph representation for programs.
- We verify the necessity of studying graph representation of Kotlin separately by comparing the difference between Kotlin and Java in graph representation through the method of induction and summary.
- Through a downstream source-code query task, we proved our graph-based method is more effective than text-based methods. We also proved that language features are useful to enhance graph representation with no greater cost.

II. BACKGROUND

To our best knowledge, there has not been any study on the graph representation method of Kotlin Language. Yet, some Kotlin related studies and graph representation of

* Corresponding author
DOI reference number: 10.18293/SEKE2022-079

program do exist. In this section, we will review the relevant literature.

A. Kotlin

Since Kotlin has 100% interoperability with Java, quite a few Kotlin related studies did a comparative study between Java and Kotlin language[7][8][9]. They concluded that Kotlin is concise and safe, can improve the productivity of the programmer, and also improve the quality of applications.

Besides, there are several empirical studies conducted on its adoption by the developers. Mateus.B.G et al.[10] explored the source code of 387 Android applications, describes the evolution of usage of Kotlin language features on these applications. Martinez.M et al.[11] and Coppola.R et al.[12] did research on the evolution of Java to Kotlin and believed that Kotlin can ensure the seamless migration of Android developers from Java to Kotlin.

The current research on Kotlin mostly stays at analyzing the differences between Kotlin and Java in terms of syntax, performance, language features, as well as the research on the quality and performance of programs written in Kotlin, the study of graph representation of Kotlin is still empty.

B. Graph Representation of source code

There is currently no widely accepted method for mapping programs into graph representation, different studies have different graph representation methods. Allamanis et al.[3] mapped program to graphs consisting of ASTs together with control-flow edges, data-flow edges, and a hand-crafted set of additional typed edges. Lu M et al.[4] proposed a program graph method named FDA, which integrates the AST, function call graph, and data-flow graph to characterize syntax and semantic information. T. T. Nguyen et al.[5] proposed a program graph method named Groum, where nodes represent actions and control points, and edges represent control and data flow dependencies between nodes.

Almost all graph representation methods are coupled to language, like Allamanis et al.'s work[3] is for C#, Lu M et al.'s work[4] is for C++ and Nguyen et al.'s work[5] is for Java. But Kotlin has no related graph representation methods. This is mainly due to the fact that Kotlin is a relatively new language and there is a lack of tools and tagged datasets for further study.

III. ON THE NECESSITY OF STUDYING KOTLIN SEPARATELY

As Java is a mature programming language and Kotlin is so connected with Java, can we directly convert Kotlin to Java, and then map the converted code to graph representation? To verify the necessity of studying Kotlin separately, we compare Kotlin and Java in terms of 3 dimension in this section and found some profound differences that prevent directly using graph representation methods of Java to map Kotlin code.

A. Decompile Kotlin Code to Java Code

Kotlin is 100% interoperable with Java. Kotlin and Java source code can even be converted to each other. Tools in IntelliJ IDEA and Android Studio can help us do such conversion. But there are some problems:

1) Kotlin program must be compiled before decompiled. But it's difficult to successfully compile each projects, for there are always many environment and configuration requirements.

2) There is currently no tool for batch decompilation. We need to decompile Kotlin code file by file if we want to decompile the whole project.

3) The conversion effect is unsatisfactory. The Java file decompiled from Kotlin always has many redundant meaningless encoding and some even have bugs. This also implies that the conciseness of Kotlin makes it lose some information, and it may be harder to analyze kotlin

To sum up, decompiling Kotlin code into Java code is time-consuming and troublesome.

B. Kotlin's Language Features

Kotlin provides programmers with various language features that make it concise, safe and expressive. We collected 30 Kotlin language features as Table I summarized. These features are further summarized on the basis of Mateus B G's work[10]. They extracted 24 Kotlin features from a document that compares Kotlin and Java[13], Kotlin's releases notes, and Kotlin Reference[14]. We inspect these documents and websites, update 6 new features (marked in the table) in our list.

TABLE I. LANGUAGE FEATURES IN KOTLIN

id	feature	id	feature
1	Type inferences	16	singletons
2	Lambda expressions	17	Companion object
3	Inline function	18	Destructing declaration
4	Null-safety	19	Infix function
5	When expressions	20	Tail-recursive function
6	Function w/arguments with a default value	21	Sealed class and sealed interfaces
7	Function w/named arguments	22	Type aliases
8	Smart casts	23	coroutines
9	Data classes	24	contract
10	Range expressions	25	Inline classes
11	Extension functions	26	properties
12	String template	27	Primary constructors
13	First-class delegation	28	Operator overloading
14	Declaration-site variance & Type projections	29	Separate interfaces for read-only and mutable collections
15	Suspending functions	30	Instantiation of annotation classes

We can see that there are plenty of language features that exist in Kotlin but not present in Java. If we convert Kotlin to Java to construct graph representation, we will lose these Kotlin language features in the graph, which contain lots of information and represent Kotlin's characteristics.

C. Verbosity vs Concise

Java is a verbose language, yet Kotlin's syntax focuses on removing verbosity. Rough estimates indicate approximately a 40% cut in the number of LOC compared to Java[8]. Fig. 1 is a Kotlin code snippet and its decompiled Java code. Java code is nearly 3 times longer than Kotlin to implement the same function. Therefore, converting Kotlin to Java to construct graph representation would take away such concise

characteristics in Kotlin, which is one of the most significant features of Kotlin.

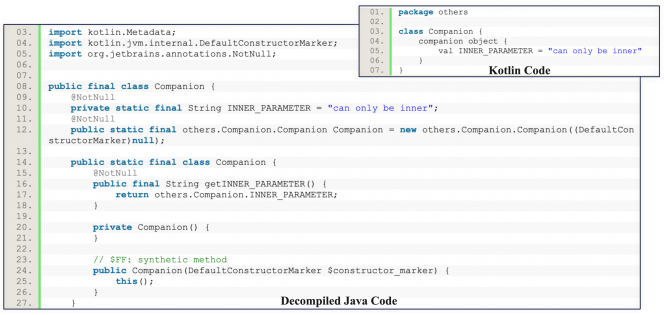

Figure 1. Kotlin code and its decompiled Java code

In conclusion, constructing graph by converting Kotlin to Java is not only troublesome; but also will lose Kotlin's crucial features. Therefore, it is necessary to study the graph representation of Kotlin separately.

IV. GRAPH MAPPING STRATEGY

A. Graph Representation of Code

We use composite code representation and denote Kotlin code by a joint graph with three types of sub-graphs (AST, CFG, and DFG) and enhanced by language features nodes, edge, and attributes. All edges in the graph are directed edges.

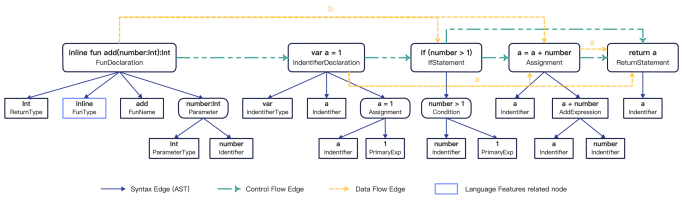

Figure 2. Graph representation example

AST (Abstract Syntax Tree) is the fundamental structure of program and it contains almost all syntax information of program, we use AST as the backbone of the graph representation of Kotlin. The major AST nodes are shown in Fig. 2. All boxes are AST nodes, with specific codes in the first line and node type annotated. The dark blue arrows represent the child-parent AST relations, which are called syntax edges in our methods.

CFG (Control Flow Graph) describes all paths that might be traversed through a program during its execution. The path alternatives are determined by conditional statements, e.g., if, for, and switch statements. In CFGs, nodes are connected by directed edges to indicate the transfer of control. The CFG edges are highlighted with green arrows in Fig. 2. Two different paths derive from the if statement.

DFG (Data Flow Graph) tracks the usage of variables throughout the CFG. Data flow is variable oriented and any data flow involves the access or modification of certain variables. A DFG edge represents the subsequent access or modification onto the same variables. It is shown by yellow arrows in Fig. 2 with the involved variables annotated over the edge. For example, the identifier a is used both in the assignment statement and the return statement.

Language Features extract common patterns in source code, containing much information. We explicitly represent Kotlin language features in the graph, some by adding edges, some by adding nodes, and some by adding attributes to nodes, according to each features' characteristics. In Fig. 2, inline function feature is represented by a node with function type inline that is illustrated by the light blue box.

B. Construction Process

The construction process is shown in Fig. 3, we first use *kotlinx.ast*, a generic AST parsing library, to extract functional external AST. This library only can parse ASTs outside functions, so we need to manually extract ASTs inside functions by traversing code, which is carried out to the statement level. Nodes in ASTs represent statements, and they are classified into 19 types according to their grammatical structure information, such as *IdentifierDeclaration*, *IfStatement*, etc. And we connected these nodes by syntax edges according to their syntactic relationship.

Then, We extract control flow by analyzing ASTs we have already gotten, including syntax information of ASTs and control statements. Then we record the scope of variables and statements that use the variable to get data flow.

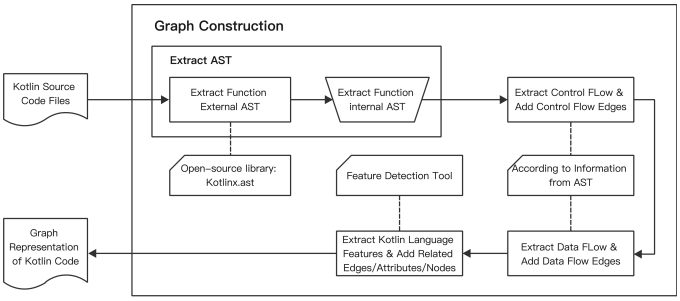

Figure 3. Graph construction process

To extract Kotlin language features, we built a feature detection tool operating on Kotlin source code and ASTs. For each language features presented in Table I, we first manually investigated how a feature is represented in source code. Then we encoded different analyzers for detecting feature instances on source code files. For features that cannot be detected in source code, we extracted them by analyzing raw ASTs. Then, we add these features information into the joint graph.

C. Points to Note

Some points need to note in the graph construction process, including problems, limitations, and some hints.

Lack of tools Kotlin is a relatively new programming language and there is a lack of tools, which creates difficulties for Kotlin code analysis. For example, there are only two AST parsing libraries for AST, *kotlinx.ast*[2] and *kastree*[3]. But both of them are with limited function and

[2] "kotlinx/ast", https://github.com/kotlinx/ast, 2022-03-11.
[3] "kastree", https://github.com/cretz/kastree, 2022-03-11.

cannot satisfy our requirements. Therefore, if one needs to generate Kotlin graph representation in large batches, you should develop a robust Kotlin AST parser first.

Maintain a Kotlin language feature diagram Kotlin is an actively released language, new language features will be introduced in future releases. In order to ensure the integrity of the language features that can be represented in the graph, one should maintain a Kotlin language feature form. Every time there is a new release, one should check the release note and keep the diagram up to date.

Feature Representation Method Different language features have different representation methods including adding edge, adding nodes, and adding attributes to nodes, according to their characteristics. Each features need to be investigated separately to decide how to represent it in the graph is more reasonable and analysis-friendly.

V. EXPERIMENT

In this section, we describe our experiment and discuss the result. Our experiment is guided by the following research questions:

- RQ1: How our graph-based method performs comparing to text-based methods?
- RQ2: Whether is it effective to add language features to graph representation?

A. Experiment Framework for Down-stream Task

Software developers and tools often query source code to explore a system, or to search for code for maintenance tasks such as bug fixing, refactoring, and optimization. Considering the objectivity of the downstream task, we choose Wiggle[15], a representative source code query system based on the graph data model as a reference for our downstream tasks. We transform it to support Kotlin query, forming a code query framework for our experiment as shown in Fig. 4.

We develop a graph representation constructor as we describe in Section IV, mapping Kotlin source code to graph representation. Then we store graph representation into a Neo4j graph database. For a given query, the framework would return the code excerpts found (labeled with their location).

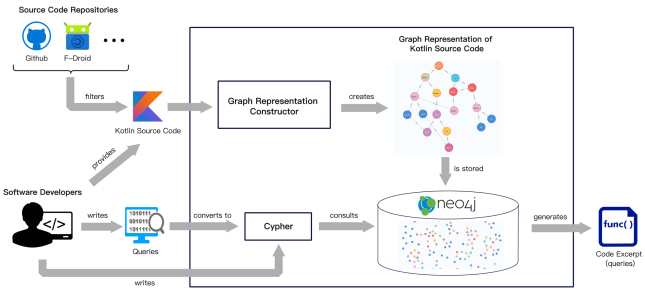

Figure 4. Source code query framework

B. Experiment Setup

1) Basic Setup

We executed various queries for different scenarios on a corpus of three Kotlin Projects on GitHub (shown in Table II). The dataset is relatively small because of the lack of tools as we noted in Section IV, limiting us generate graph representation in a large batch. But the data volume of this size is sufficient for our study. After converting all source code into graph representation and storing them into Neo4j, there are 2462 nodes and 2556 edges (relationships) in the database.

TABLE II. DATASET

Dataset	Description	LoC
Kotlin101	A collection of runnable console applications that highlights the features of Kotlin.	747
KAndroid	Kotlin library for Android to eliminate boilerplate code in Android SDK and focus on productivity.	886
Android-SearchView	A demonstration application for android's SeachView.	415

All the tests commence on a MacBook Pro with 8-core CPU, 16GB unified memory, and 512GB SSD. The graph database Neo4j Browser version is 4.4.2 and Server version is 4.3.10 (community).

2) Query Selection

We select 11 queries in three query scenarios, which consist of language research, complex search, and program check. Language research and complex search refer to Wiggle's query examples[15] and program check refer to the evaluation part in Rodriguez-Prieto O et al.'s work[16]. We modified queries in their work, forming 11 queries shown in Table III.

TABLE III. SAMPLE QUERIES

id	Purpose	Query
Q1	Language Research	lambda expression
Q2		companion object
Q3		for statement
Q4		Inline function
Q5	Complex Search	function with *Int* return type
Q6		search for classes containing recursive methods
Q7		find instances of classes that inherited from *People*
Q8	Program Check	Binary conditions prefer *if* over *when*
Q9		Public functions/methods that return platform type expressions must explicitly declare their Kotlin type
Q10		Return an empty array or collection instead of a null value for methods that return an array or collection
Q11		Convert integers to floating point for floating-point operations

C. Result and Discussion

- **RQ1: How the graph-based method performs compare to text-based methods?**

We provide a comparative study of text-based methods and our graph-based method to answer this question. We choose keyword match and regular expressions as text-based source code search approach in our comparative study, which is the most common and widely used text-based approaches for searching code for now.

First, we evaluate the coverage of these three approaches. The result is shown in Table IV. Obviously, graph-based method can cover more queries than the other two text-based approaches, especially more complex search including more dependency and requirements. This is mainly because that code is texts with structures and semantics, such information is implicit that these text-based search approaches are unable to capture. In contrast, graph representation of code contains plentiful syntax and semantic information, provided by nodes and their relations, which can be leveraged for effective search.

TABLE IV. COVERAGE OF DIFFERENT APPROACHES

Query_id	keyword	Regular expression	Graph representation
Q1		√	√
Q2	√	√	√
Q3	√	√	√
Q4	√	√	√
Q5	√	√	√
Q6			√
Q7			√
Q8		√	√
Q9			√
Q10	√	√	√
Q11			√

Then, for queries that text-based methods and graph-based method all can cover, we conduct further evaluation by introducing two performance indicators. HitRate is the percentage of correct search results out of all correct results, evaluating the exhaustion of search approaches. P@all evaluate the precision of search results. It calculates by the percentage of correct search results out of all search results. The result is shown in Table V.

TABLE V. TEST RESULT

id	HitRate			P@all		
	Key word	*Regular expression*	*Graph-based*	*Key word*	*Regular expression*	*Graph-based*
Kotlin101						
Q1	N/A	20%	100%	N/A	60%	100%
Q2	100%	100%	100%	70%	50%	100%
Q3	100%	100%	100%	67%	100%	100%
Q4	100%	100%	100%	70%	50%	100%
Q5	100%	100%	100%	17%	100%	100%
Q8	N/A	100%	100%	N/A	100%	100%
Q10	100%	100%	100%	25%	67%	100%
KAndroid						
Q1	N/A	26%	100%	N/A	100%	100%
Q2	-	-	-	-	-	-
Q3	100%	100%	100%	4%	100%	100%
Q4	100%	100%	100%	93%	100%	100%
Q5	100%	100%	100%	10%	80%	100%
Q8	N/A	100%	100%	N/A	100%	100%
Q10	0%	100%	100%	0%	20%	100%
Android-SearchView-Demo						
Q1	N/A	0%	100%	N/A	0%	100%
Q2	100%	100%	100%	100%	100%	100%
Q3	100%	100%	100%	40%	67%	100%
Q4	-	-	-	-	-	-
Q5	100%	100%	100%	91%	100%	100%
Q8	N/A	100%	100%	N/A	100%	100%
Q10	100%	100%	100%	33%	50%	100%

For both HitRate and P@all, Graph-based approach has significantly better performance, especially for P@all. Text-based approaches has low P@all, because they often return some code that matches the query but is not related, like description in comment or unrelated code contains keyword or match the regular expression. Fig. 5 shows some unrelated results examples with their corresponding queries in text-based methods for Q5. Better performance of graph-based methods shows that preferentially extracting language features and add them to graph representation is more promising than extracting them directly from text.

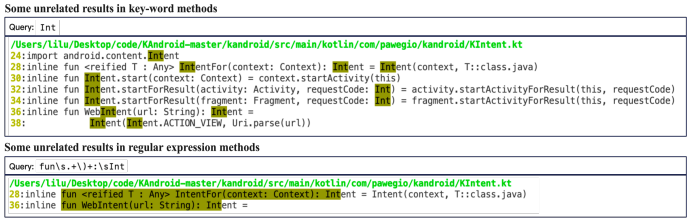

Figure 5. Unrelated results examples in text-based methods

- **RQ2: Whether is it effective to add language features to graph representation?**

To assess the coverage of our approach, we analyze which code representations are necessary to describe different kinds of queries. The results of this analysis are presented in Table VI.

Obviously, the AST alone provides only a little information for querying source code. By combining AST with CFG or DFG, we obtain a better view of the code and can describe almost every query except language feature-related query. But language features contain much information and represent characteristics of Kotlin, so language features related queries should not be excluded. After adding language features, we are finally able to model all the query samples, making use of information available from AST, CFG, DFG, and language features representation.

TABLE VI. COVERAGE OF DIFFERENT GRAPH REPRESENTATION

id	AST	AST+ CFG	AST+ DFG	AST+language features	AST+CFG+ DFG+language features
Q1				√	√
Q2				√	√
Q3	√	√	√	√	√
Q4				√	√
Q5	√	√	√	√	√
Q6		√			√
Q7		√			√
Q8	√	√	√	√	√
Q9	√	√	√	√	√
Q10	√	√	√	√	√
Q11			√		√

We also calculated the graph complexity of different graph representations as shown in Table VII. The graph complexity measurement method uses the graph measures proposed by Dehmer M et al[17] using a polynomial-based approach. Through the comparison, we can see that adding language

features in the graph representation will not increase the complexity greatly, but it really can represent more information.

TABLE VII. COMPLEXITY OF DIFFERENT REPRESENTATION

Dataset	AST	AST+CFG+DFG	AST+CFG+DFG+ language features
Kotlin101	0.28351	0.35987	0.37066
KAndroid	0.16175	0.20886	0.21384
SearchView	0.36623	0.41296	0.42089
Average	0.27049	0.32723	0.33513

VI. THREAT TO VALIDITY

ASTs inside functions. Because of the lack of tools, we need to manually extract ASTs insides functions by using types of nodes and edges defined by ourselves. So ASTs inside functions may lack of unity. When nodes and edges are defined differently, the results are different.

Feature representation method. Different language features have different representation methods including adding edges, adding nodes, and adding attributes to nodes, according to their characteristics. Feature representation in this paper is all reasonable, but anyway, there will be better ways of representation, which also will affect the results.

Queries of text-based methods. Though key-word method and regular expression method is standardized, queries is not. Different queries will lead to different results.

VII. CONCLUSION

In this paper, we conduct a pilot study on graph representation method of Kotlin source code. We first verify the necessity of studying graph representation of Kotlin separately by comparing the difference between Kotlin and Java in graph representation through the method of induction and summary. Then we proposed a first-sight, rule-based graph mapping method. It takes AST as the skeleton, enriching with control flow edge and data flow edge, together with some edges and attributes representing Kotlin's language features ostensive. We present our graph construction process and summarized points to note in the process, aiming to provide other Kotlin downstream task researchers with a basis for constructing graph representation for programs.

We evaluate our method through a source-code query down-stream task, came to the following conclusions: 1) Graph representation methods outperform text-based methods both on query coverage and search result. This is because graph representation contains more syntax and semantic information which can be well leveraged in source-code search and even other source-code analysis tasks. 2) Language features are useful to enhance graph representation. First, graph representation with language features can cover more queries. In addition, preferentially extracting language features and adding them to graph representation is more promising than extracting them directly from text. 3) Adding language features to graph representation will not add much complexity.

In the future, we plan to study next steps of code analysis for Kotlin, including graph embedding, neural network, and so on, aiming to conduct a full-link study on code analysis for Kotlin and promote its application in the field of “big code”. Furthermore, our study with Kotlin is instructive for the analysis of similar concise modern languages, that adding language features to graph representation is an exploration direction.

REFERENCES

[1] “The State of the Octoverse 2018,” https://octoverse.github.com/2018/projects#languages, last access: 2022-03-11.

[2] “Top Programming Languages 2021,” https://spectrum.ieee.org/top-programming-languages/, last access: 2022-03-11.

[3] Allamanis M, Brockschmidt M, Khademi M. Learning to represent programs with graphs[J]. arXiv preprint arXiv:1711.00740, 2017.

[4] Lu M , Tan D , N Xiong, et al. Program Classification Using Gated Graph Attention Neural Network for Online Programming Service[J]. 2019.

[5] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. AI-Kofahi and T. N. Nguyen, "Graph-based mining of multiple object usage patterns", Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering ser. ESEC/FSE '09, pp. 383-392, 2009.

[6] Allamanis M. Graph Neural Networks in Program Analysis[M]//Graph Neural Networks: Foundations, Frontiers, and Applications. Springer, Singapore, 2022: 483-49

[7] Gotseva D, Tomov Y, Danov P. Comparative study Java vs Kotlin[C]//2021 27th National Conference with International Participation (TELECOM). IEEE, 2021: 86-89.

[8] Flauzino M, Veríssimo J, Terra R, et al.. Are you still smelling it? A comparative study between Java and Kotlin language[C]//Proceedings of the VII Brazilian symposium on software components, architectures, and reuse. 2018: 23-32.

[9] Mateus B G, Martinez M. An empirical study on quality of Android applications written in Kotlin language[J]. Empirical Software Engineering, 2019, 24(6): 3356-3393.

[10] Mateus B G, Martinez M. On the adoption, usage and evolution of Kotlin features in Android development[C]//Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). 2020: 1-12.

[11] Martinez M, Mateus B G. How and Why did developers migrate Android Applications from Java to Kotlin? A study based on code analysis and interviews with developers[J]. arXiv preprint arXiv:2003.12730, 2021.

[12] Coppola R, Ardito L, Torchiano M. Characterizing the transition to Kotlin of Android apps: a study on F-Droid, Play Store, and GitHub[C]//Proceedings of the 3rd ACM SIGSOFT International Workshop on App Market Analytics. 2021: 8-14.

[13] 2016. Comparison to Java Programming Language. https://Kotlinlang.org/docs/ reference/comparison-to-Java.html Online; accessed 01-July-2019.

[14] JetBrains. 2019. Kotlin Language Documentation. https://Kotlinlang.org/docs/Kotlin-docs.pdf.

[15] Urma R G, Mycroft A. Source-code queries with graph databases—with application to programming language usage and evolution[J]. Science of Computer Programming, 2015, 97: 127-134.

[16] Rodriguez-Prieto O, Mycroft A, Ortin F. An efficient and scalable platform for Java source code analysis using overlaid graph representations[J]. IEEE Access, 2020, 8: 72239-72260.

[17] Dehmer M, Chen Z, Emmert-Streib F, et al. Measuring the complexity of directed graphs: A polynomial-based approach[J]. Plos one, 2019, 14(11): e0223