INTEGRATING DECLARATIVE STATIC ANALYSIS WITH NEURAL MODELS OF CODE

Pardis Pashakhanloo

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2022

Supervisor of Dissertation

Mayur Naik, Professor, Computer and Information Science

Graduate Group Chairperson

Mayur Naik, Professor, Computer and Information Science

Dissertation Committee

Osbert Bastani, Assistant Professor, Computer and Information Science
Boon Thau Loo, Professor, Computer and Information Science
Insup Lee, Professor, Computer and Information Science
Petros Maniatis, Staff Research Scientist, Google Brain

INTEGRATING DECLARATIVE STATIC ANALYSIS
WITH NEURAL MODELS OF CODE

Dedicated to my biggest fans:
mom and dad.

ACKNOWLEDGEMENT

In the first place, I would like to extend my deepest gratitude to my advisor, Mayur Naik, for providing me with invaluable insight, guidance, and support throughout my Ph.D. journey. Among other lessons, he taught me to never lose sight of the bigger picture in my life and career. A crucial advice I learned from him in research was that solving one great problem is more valuable than solving multiple superficial ones. I would not have been able to succeed in this endeavor without his mentorship. Also, I wish to express my sincere gratitude to Boon Thau Loo, whose advice and support provided me with the courage to overcome many obstacles along the way.

I wish to thank my committee members, Petros Maniatis, Osbert Bastani, Boon Thau Loo, and Insup Lee, whose constructive feedback significantly improved this dissertation. I would like to especially thank Petros for offering insightful advice and suggestions on research, communication, and professional development, in addition to being an invaluable member of my Ph.D. committee.

Many thanks to my collaborators, particularly Hanjun Dai, Yuepeng Wang, and Aaditya Naik, with whom I had several discussions that ultimately led to this dissertation.

I wish to acknowledge the contributions of the department chair, Zack Ives, the associate dean, Boon Thau Loo, the graduate chair, Mayur Naik, and the associate chair, Steve Zdancewic, to the improvement of the computer and information science department.

I would like to thank my friends who were like family to me during these years: Farhad Pashakhanloo, Mehrafshan Jafari, Armaghan Fakhraei, Sara Eskandari, and Ian Dardani. They made Philadelphia my home away from home with their warmth and kindness.

It is my profound privilege to have parents who unconditionally loved and supported me during every chapter of my life. I am forever indebted to them for their continuous support, words of encouragement and motivation, and insightful suggestions in my highest and lowest moments. It is impossible to express my gratitude to them in words.

ABSTRACT

INTEGRATING DECLARATIVE STATIC ANALYSIS WITH NEURAL MODELS OF CODE

Pardis Pashakhanloo

Mayur Naik

In recent years, deep learning techniques have made remarkable strides in solving a variety of program understanding challenges. The successful application of these techniques to a given task depends heavily on how the source code is represented by the deep neural network. Designing a suitable representation for a newly created task involves many challenges. It is necessary, among other things, to understand the implementation of other functions or modules in a project that may be spread out across a large lexical area. In addition, determining which components and features to include in order to enrich the representation is a challenge. In this dissertation, the challenges of code representation are addressed by proposing to systematically represent programs as relational databases, introducing a graph walk mechanism to remove unrelated context from large relational graphs, and describing a language for specifying tasks and program analysis queries to tailor neural code-reasoning models. A detailed analysis shows the presented techniques are superior to state-of-the-art in a variety of aspects, such as performance, robustness, and interpretability.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF ILLUSTRATIONS

CHAPTER 1

Introduction

Deep learning techniques have made significant strides in solving a wide range of program understanding tasks (Lu et al., 2021). They have been applied to code-reasoning tasks, including bug detection (Pradel and Sen, 2018; Allamanis et al., 2021; Shi et al., 2021; Dinella et al., 2020; Allamanis et al., 2018), type inference (Hellendoorn et al., 2018), code summarization (Wang et al., 2020; Ahmad et al., 2020; Ma et al., 2022; Gao and Lyu, 2022), program repair (Vasic et al., 2019; Allamanis et al., 2021; Li et al., 2020; Dinella et al., 2020), and code generation (Li et al., 2022; Alon et al., 2019a; Ottens et al.; Svyatkovskiy et al., 2020; Mukherjee et al., 2021), among many others.

The successful application of these techniques to a given task depends heavily on how the source code is represented by the deep neural network. Designing a suitable representation for a new task involves making many crucial choices. It is necessary, for instance, to understand the implementation of other functions and modules within a project in order to make informed predictions about certain tasks. Thus, the desirable context for a model may extend beyond the local lexical neighborhood, possibly requiring reasoning about a chain of called functions.

Another crucial choice is deciding which program elements and features to include when representing programs. Programs are usually represented as graphs such as abstract syntax trees (Alon et al., 2018, 2019b), control flow graphs (Allamanis et al., 2018), or dataflow graphs (Guo et al., 2020a). To achieve graphs that are richer in semantic information, others have proposed custom, non-trivial graphs that include hand-engineered lexical, syntactic, and semantic edges (Allamanis et al., 2018; Hellendoorn et al., 2020). It has been shown that composite graphs can indeed yield promising results (Siow et al., 2022). Despite this, existing approaches do not combine information in a systematic and extensible manner.

Richness of information and extended scope imply that the relevant context may be very

broad. Models tackling such tasks are challenged to either reduce the scope of a task—e.g., a single function, or a few contiguous lines of code text—or heuristically sample from a larger scope to produce a small enough input to fit inside the memory of a GPU. For example, Transformers (Vaswani et al., 2017) learn to reason about code from a sequence of tokens in the program; and GNNs (Allamanis et al., 2018) with n layers—a hyperparameter which, for message-passing architectures, determines how much of the graph is *reachable* from some immediately adjacent context to the task instance—prune all nodes that are further than n graph hops away. Despite being an important challenge, distance is not always the determining factor in collecting relevant information. As an example, when deciding what exception type is applicable to a particular part of the source code, following control flow edges until they encounter a raised exception may be more critical than fetching all adjacent statements that do not handle exceptions.

The use of token sequences to represent programs (Kanade et al., 2020; Feng et al., 2020; Chen et al., 2021) bypasses the difficulties of hand-engineered program graphs, but hampers the robustness and interpretability of these models (Xu et al., 2022). Despite the seemingly intelligent predictions that these models make, their objective is usually to reduce the error with respect to predicting a label or a masked token. Ultimately, developers are responsible for making the final decisions. So, models should be able to provide useful information in addition to the final prediction (Lipton, 2018)—however, these models do not provide such information. In addition, representing programs using token sequences has shown to be sensitive to noise and adversarial attacks (Zeng et al., 2022; Bielik and Vechev, 2020; Henke et al., 2022; Wang et al., 2022).

Another possible shortcoming of representing programs as token sequences is that they find it difficult to capture code syntax and semantics[1] especially when a small amount of data is available. According to Siow et al. (2022), token sequence representation of source code is inferior to other methods such as tree- or graph-based approaches in detecting vulnerabilities

[1]This viewpoint is controversial and still undergoing research and investigation.

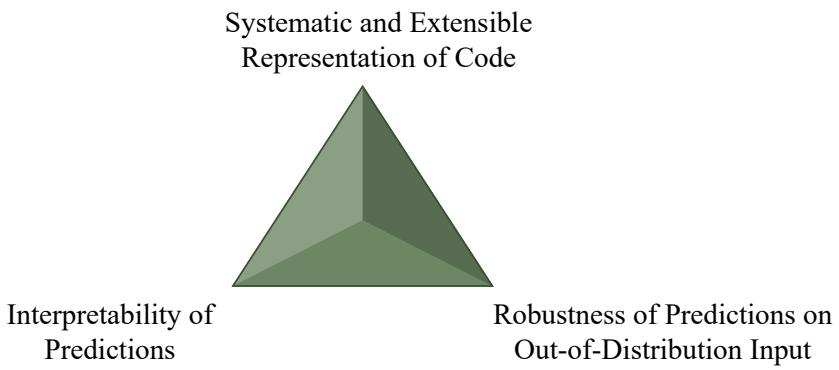

Figure 1.1: Dissertation Goals.

and classifying programs.

To overcome the mentioned difficulties, I build upon recent advances in program analysis that represent codebases as semantically rich relational databases. Throughout this dissertation, I demonstrate how the knowledge locked up in decades of program analysis research can be an invaluable resource for building models of code that are more accurate, more robust, and inherently interpretable.

In this dissertation, I present approaches to achieve the goals that are represented in Figure 1.1. I achieve **Systematic and Extensible Representation of Code** through a hybrid approach that combines the logical analysis done by a code query engine with the statistical analysis done by machine learning. Especially since collecting labeled data for code-understanding tasks is difficult, one must utilize the data effectively during the training process (Le et al., 2020). My proposed approach, called CodeTrek, offloads the engineering cost of extracting semantic information from programs to a standard tool in a systematic way. CodeTrek transforms the relational database of source code into a relational graph using key-foreign-key relationships. The resulting semantically rich representation allows the model to focus on solving more complicated tasks instead of struggling to learn the information that is already computed by simple program analyses. CodeTrek also allows developers to extend the representation by writing code queries in a code query language called CodeQL (Avgustinov et al., 2016). CodeTrek samples guided random walks over

relational program graphs depending on the task and embeds them. This enables the **Robustness of Predictions on Out-of-Distribution Input**. I further introduce a technique to learn to walk over large program graphs to obtain high accuracy on any new code understanding task. CodeTrek also allows **Interpretability of Predictions**. The walks that contribute the most to the final predictions serve as a witness or explanations, making CodeTrek intrinsically interpretable.

In summary, my dissertation contributes the following to the field.

Developing the Foundations for a Relational Representation of Code

I propose to represent programs as relational databases that make rich context readily available for code-reasoning tasks using deep learning. I present an effective algorithm to construct graphs from relational representations of code. Also, I present a graph-walk mechanism that prunes unrelated context in a task-specific manner. To define new or existing code-reasoning tasks, I propose a specification language. This specification language allows us to effectively direct the generation of graphs and walks.

Introducing Techniques to Enhance Bug Finding using Neural Models

I present a technique to learn walk policies that prune large relational graphs by ranking the relations based on their relevance in a task-specific manner. Using program analysis queries and systematic test-program generation, I propose techniques to enable task designers to stress-test their models. I identify two new challenging tasks for neural code reasoning, *unused definition* and *variable shadowing*; although sophisticated, non-neural static-analysis tools can solve them, these tasks pose a useful litmus test for neural code-reasoning frameworks and demonstrate the ability of CodeTrek to generate challenging tasks that follow real-world program distributions with modest effort. In my extensive evaluation, I demonstrate that deeper relational information about code helps neural models outperform the state-of-the-art.

Demonstrating the Superiority of Relational Representation of Code in Robustness and Interpretability

I instantiate CodeTrek for code-to-text translation. Using an extensive evaluation, I demonstrate CodeTrek's superiority over the state-of-the-art in terms of accuracy and robustness against semantic-preserving code transformations. In addition, I conduct a qualitative study of CodeTrek's intrinsic interpretability.

CHAPTER 2

Background

In this chapter, I describe neural network architectures (Section 2.1) including Graph Neural Networks (Section 2.1.1), Transformers (Section 2.1.2), and Deep Sets (Section 2.1.3). Afterwards, I introduce existing methods for generating text (Section 2.2) and neural network embeddings (Section 2.3). I conclude the chapter by reviewing code databases (Section 2.4).

2.1. Neural Network Architectures

Deep learning refers to a type of machine learning that allows learning from a hierarchy of concepts. This approach avoids the need for domain experts to formally specify each feature that the machine learning algorithm needs. It allows computers to learn complex concepts on top of simpler ones (Goodfellow et al., 2016). Modern deep learning provides a powerful framework for supervised learning (Goodfellow et al., 2016). Here, I provide an overview of the deep neural network architectures discussed in this dissertation.

2.1.1. Graph Neural Networks

A graph is a flexible data structure that represents the relationships between a collection of nodes. Graph Neural Network (GNN) is a framework for defining deep neural networks over graphs. The key idea is to generate representations of nodes based on the structure of the graph and its entities (Hamilton, 2020). The defining feature of a GNN is that it uses neural message-passing in which vector messages are exchanged between nodes and updated using neural networks. A GNN is composed of different iterations of message-passing, also known as layers.

The basic intuition behind the GNN message-passing framework is that, each node contains some information at the beginning. At every message-passing iteration, each node aggregates the information from its local neighborhood with its current information to obtain updated information. So, after every iteration, the information that each node carries is updated (Geerts et al., 2021). The resulting node information is called node embedding.

After the first iteration ($k = 1$), every node embedding contains information about its 1-hop neighborhood, i.e., every node embedding contains information about the features of its immediate graph neighbors, which can be reached by a path of length 1 in the graph; after the second iteration ($k = 2$) every node embedding contains information about its 2-hop neighborhood; and, after k iterations every node embedding contains information about its k-hop neighborhood (Hamilton, 2020).

Node embeddings capture different types of graph information. The first kind is structural information. This kind of information is useful when predictions depend on the overall shape of the neighborhood of nodes. Another kind of information that the embedding captures is the content, or features, of the nodes. GNN captures this kind of information based on local graph neighborhoods (Hamilton, 2020).

Mathematically, the message-passing framework can be described in terms of *update* and *aggregate* operations:

$$h_u^{k+1} = update^k\Big(h_u^k, aggregate^k(h_{v\in\mathcal{N}(u)}^k)\Big) \tag{2.1}$$

In Equation 2.1, $\mathcal{N}(u)$ is the set of immediate neighbors of node u, h_u^k is the embedding of node u at iteration k, and *aggregate* and *update* are arbitrary differentiable functions. Sometimes, $aggregate^k(h_{v\in\mathcal{N}(u)}^k)$ is called the *message*. After K iterations, the output of the final layer produces the final embedding for each node. Varying *update* and *aggregate* functions results in different flavors of GNNs. Some of these variations are shown in Table 2.1. Prominent examples include Graph Convolutional Networks (GCN) (Welling and Kipf, 2016), GraphSAGE (Hamilton et al., 2017), and Graph Attention Networks (GAT) (Veličković et al., 2018). In GCN, the aggregate function is an average function, and the update function is a non-linear activation function, like ReLU. GraphSAGE uses a fully connected neural network for updates and an LSTM for aggregates. Finally, GAT computes a weighted average of neighboring node features using an attention module, and updates the features using a summation function and a non-linear activation function.

Table 2.1: Variations of *update* and *aggregate* functions.

update	***aggregate***
Average	Average
Maximum	Maximum
Summation	Summation
Neural Network	Neural Network

2.1.2. Transformers

Sequence-to-Sequence models are neural networks that transform input token sequences into output token sequences. These models are often used for translating from one language to another (e.g., Python to Java or English to French).

Overall Architecture. Transformers (Vaswani et al., 2017) are sequence-to-sequence neural networks that are effective at natural language processing. The Transformer architecture, like some other sequence-to-sequence models, consists of two key components: an *encoder* and a *decoder*. An encoder converts the input sequence into a continuous representation, which is then passed along to a decoder. The decoder receives the output of the encoder in conjunction with the output of the decoder for the previous timestep in order to generate output tokens.

Positional Encoding. In some sequences, such as source code or text documents, the order of the tokens is crucial for comprehension. So, an instrumental component of the Transformer model is a *positional encoding*. Positional encoding adds relative position information to every token in a sequence. Suppose there is an input sequence of length L. Equation 2.2 shows how the positional encoding of the kth element is computed using sine and cosine functions. In this equation, $k \in [0, L/2)$ is the position of the element in the sequence, d is the dimension of the output embedding space, and n is a user-defined hyperparameter that is set to 10000 by Vaswani et al. (2017).

$$P(k, 2i) = sin(\frac{k}{n^{2i/d}}), P(k, 2i+1) = cos(\frac{k}{n^{2i/d}}), 0 \leq i < d/2 \tag{2.2}$$

Attention Mechanisms. Attention computation (Bahdanau et al., 2014) is a mechanism in neural networks that selectively focuses on certain parts of input data while processing it. Attention mechanisms have become a popular tool in deep learning, particularly in natural language processing and computer vision tasks. This is because they allow the model to automatically learn which parts of the input data are most relevant for a given task.

There are several different types of attention operators. A basic type of attention is *additive attention* (Bahdanau et al., 2014), which computes a weighted sum of input values based on learned weights. It is implemented by computing a dot-product between a query vector and a set of key-value pairs, and then applying a softmax function to the dot-product to obtain attention weights. The *dot-product attention* (Vaswani et al., 2017) mechanism is similar to additive attention, but instead of computing the dot-product with learned weights, it computes the dot-product between the query and key vectors, then applies the softmax algorithm. To prevent the dot-product from becoming too large, *scaled dot-product attention* (Vaswani et al., 2017) scales the dot product by the square root of the dimension of the key and query vectors. Finally, *self-attention mechanism* (Vaswani et al., 2017) is used to compute the attention between the elements of a single input sequence. It allows the model to focus on different parts of the input sentence while processing it. I will elaborate on the concept of self-attention since it is a key concept in the following chapters.

The *self-attention mechanism* is a critical concept that helps understand Transformers and some of this dissertation's contributions. As a result of this mechanism, Transformers are able to focus on all past tokens they have generated. Self-attention *SA* is calculated using Equation 2.3. In this equation, $\mathbf{Q} \in \mathbb{R}^{d_q \times d_q}$, $\mathbf{K} \in \mathbb{R}^{d_k \times d_k}$, and $\mathbf{V} \in \mathbb{R}^{d_v \times d_v}$.

$$SA(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = Softmax(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d}})\mathbf{V} \tag{2.3}$$

Matrices in Equation 2.3 are computed as $\mathbf{Q} = \mathbf{X}\mathbf{W}^Q$, $\mathbf{K} = \mathbf{X}\mathbf{W}^K$, and $\mathbf{V} = \mathbf{X}\mathbf{W}^V$, where $\mathbf{X} = (x_1, ..., x_n)$ is the input and $x_i \in \mathbb{R}^d$. The parameter d is commonly referred to as the dimension of the hidden state. All three weight matrices $\mathbf{W}^Q \in \mathbb{R}^{d \times d_q}$, $\mathbf{W}^K \in \mathbb{R}^{d \times d_k}$, and

$\mathbf{W}^V \in \mathbb{R}^{d \times d_v}$ are learnable parameters (Vaswani et al., 2017).

2.1.3. Deep Set

The Deep Set (Zaheer et al., 2017) architecture is a neural network architecture that is designed to process sets of variable-size inputs. The key idea behind this architecture is to use a neural network to learn a function that maps a set of inputs to a fixed-size representation, which can then be used for further processing or classification. The architecture typically includes the following steps:

1. **Input Encoding:** The first step is to encode the input set into a fixed-size representation. This is typically done by applying a permutation-invariant function, such as a sum or mean, to aggregate the set of inputs into a fixed-size representation.

2. **Fully Connected Layers:** The fixed-size representation is then passed through one or more fully connected layers to extract features from the representation. These layers use weights and biases to learn the representations.

3. **Readout Layer:** The final step is to use a readout layer to make predictions based on the learned representations. This layer can be a fully connected layer or a softmax layer for classification tasks.

4. **End-to-End Training:** The model is trained end-to-end with back-propagation. The goal of training is to learn the weights and biases of the fully connected layers. This will ensure that the predictions made by the readout layer are as accurate as possible.

Deep Set architecture offers the advantage of handling variable-size inputs. This is useful in applications where the number of elements in the set can vary, but the deep set architecture can handle this by learning a fixed-size representation of the input set. Additionally, the permutation-invariant function used in the first step ensures that the architecture is robust to the order of the input elements.

2.2. Text Generation Techniques

The final step of training a model for code summarization is to assign each token in the vocabulary a value that indicates its probability of being the next token in the output. The generation of the next token continues until an end-of-sentence token is generated. There are two major techniques for generating the output: greedy decoding and beam search.

Greedy Decoding. In greedy decoding, the goal is to generate a sequence of words based on a given input. The basic idea behind greedy decoding is to generate the most likely next word at each step, based on the current input and the model's predicted probabilities. Greedy decoding starts with an empty sequence and initial input in the form of a starting token or a prompt. Then, the probability distribution of the next word over the vocabulary is computed given the current input. Afterwards, the word with the highest probability is chosen as the next word in the sequence. These steps are repeated until a stopping criterion—such as reaching a maximum length or generating an end-of-sequence token—is met. The main advantage of greedy decoding is that it is computationally efficient, as it only requires one prediction at each step. However, it can lead to repetitive and generic text, as the model only chooses the most likely next word and does not consider other possibilities.

Beam Search. An alternative strategy for text generation is beam search. In this approach, multiple sequences are built iteratively, and at the end of the search, a list of most likely output sequences is obtained. As opposed to the greedy approach, beam search expands all possible next steps and keeps the k most-likely results, where k is a parameter that controls the number of parallel searches or *beams* through the sequence of probabilities. k is also called the beam size of the beam width. Greedy search is a special case of the beam search where $k = 1$ (Goodfellow et al., 2016).

2.3. Neural Network Embeddings

In the context of neural networks, an embedding is a mapping from objects (discrete data) to vectors of real numbers (continuous space). As a result of such a transformation, one can use source code tokens or other program elements as input to neural networks (Chen and Monperrus, 2019). Neural network embeddings (Church, 2017; Alon et al., 2019b) allow the continuous vectors to preserve the similarities or dissimilarities of discrete objects. This is critical because in many applications the relationships between data points are often more relevant than the individual data points themselves. Source code can be embedded as sequences of source tokens (Kanade et al., 2020; Feng et al., 2020; Chen et al., 2021), by abstract syntax trees (Alon et al., 2018, 2019a; Dinella et al., 2020; Alon et al., 2020; Kovalenko et al., 2019), or through extensions or combinations of them (Allamanis et al., 2018; Hellendoorn et al., 2020; Guo et al., 2020a).

The reader might wonder why one-hot encoding is not used instead. A one-hot encoding approach encodes discrete variables as a bit-vector where only one bit is one or *hot*. However, this encoding has several disadvantages. First, if a variable can get many unique values, the bit-vector used for one-hot encoding can become too large. For example, the word *choice* in a vocabulary with 50K entries requires a bit-vector of size 50K. Second, one can lose essential information about the relationships between different values. In other words, the distance between two one-hot vectors does not convey any information about the distance (or similarity) of two values in the embedding space. For example, the words *choice* and *selection*, even though similar in meaning, might not be close in the one-hot encoding space.

2.4. Code Databases

The idea of querying code repositories like relational databases is intriguing. In the past few years, a number of unified frameworks for querying code have been proposed in order to assist developers in program understanding, analysis, and reverse engineering. These frameworks allow developers to make various kinds of queries. Paul and Prakash (1994) categorize code queries into two classes. The first class of queries are those that pertain to the syntax of code. Examples include fetching all the recursive functions in a codebase or retrieving classes that lack docstrings. The second class pertains to program flow and may involve inter- or intra-procedural analysis of code. Examples include computing the call graph of a codebase or finding all the uses of a particular variable.

One of the most notable examples of a query language for source code analysis is CodeQL (Avgustinov et al., 2016). It allows developers to query complex and potentially recursive data structures encoded in a relational data model (De Moor et al., 2007). CodeQL combines ideas from SQL—a declarative programming language for accessing and manipulating databases—and Datalog—a form of logic programming that has an elegant least-fixpoint semantics. The industrial-strength and optimized implementation of CodeQL—called Semmle—make it distinct from other competitors (De Moor et al., 2007). By using logic queries, Semmle helps find bugs in codebases[1]. This platform constructs databases from source code and uses CodeQL to query them. To better understand CodeQL, I explain its main components next.

CodeQL databases. A CodeQL database stores all the program information including but not limited to AST, def-use, call graph, and more. Similar to any relational database, CodeQL databases have key-foreign-key relationships, are optimized for program data, and can be queried using a SQL-like query language. Since every programming language differs in terms of its capabilities and paradigm, each requires its own extractor. For the same reasons, each programming language has its own schema for the database. The Python

[1]Semmle is adopted by GitHub (https://github.blog/2019-09-18-securing-software-together/).

schema, for instance, specifies control-flow information, dataflow-information, AST, and some meta-information. In contrast, the evaluation engine that carries out the analyses is language-agnostic.

CodeQL queries. Semmle provides a pool of pre-defined CodeQL queries for analyzing programs. As a consequence, new semantic information can be computed, such as function call graphs and point-to graphs. The fundamental approach presented in this dissertation—CodeTrek—uses the relational schema defined for Python by Semmle to construct a database for each program, and extract the tuples of each relation using CodeQL queries. An example of a CodeQL query borrowed from the official CodeQL repository is presented in Figure 2.1. This query finds functions that call themselves in Python[2].

```
1 import python
2
3 from PythonFunctionValue f
4 where f.getACall().getScope() = f.getScope()
5 select f
```

Figure 2.1: A CodeQL query to find functions that call themselves in Python.

[2]https://github.com/github/codeql/blob/main/python/ql/examples/snippets/recursion.ql

CHAPTER 3

Extensible Relational Representation of Code

In this chapter, I introduce CodeTrek (Pashakhanloo et al., 2022b), a deep learning approach that accomplishes the following key design goals. Programming languages have well-defined semantics that enable deterministic analyses to extract relevant information (such as module imports, class inheritance, inter-procedural control flow, dataflow, def-use chains, object escape, etc.). Therefore, the model can be exposed directly to it instead of learning it indirectly from labeled data. In this way, the model can focus on learning information only discovered in rich contexts. In addition, even when rich information is easily accessible, making well-informed predictions in code-reasoning tasks requires intelligent context collection to fit the needs of the task.

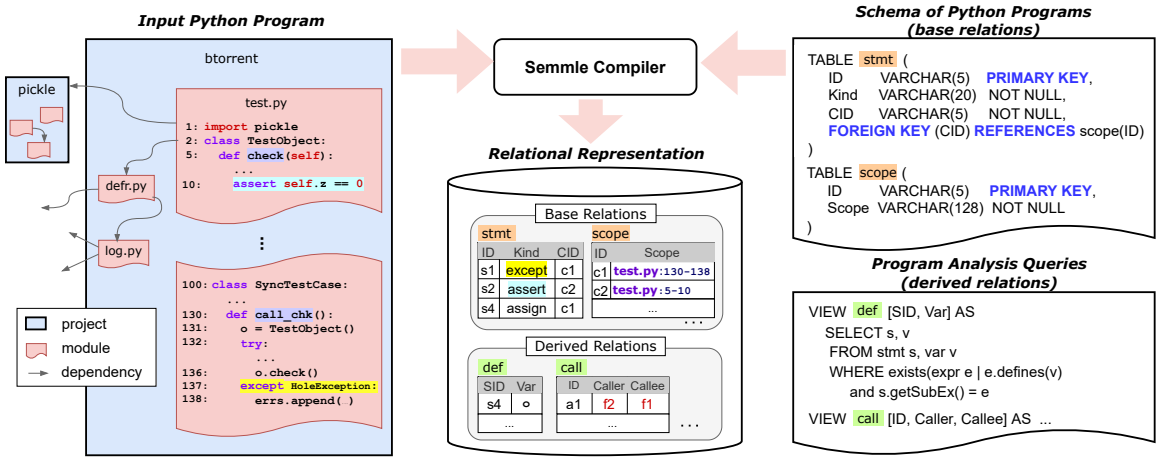

Figure 3.1: Example showing how CodeTrek translates an exception-prediction sample in a Python program into a feature-rich representation that consists of base relations that capture the program's syntax and derived relations that capture semantic information computed by program analysis queries. Specifically, CodeTrek predicts the best exception type to replace the placeholder exception type `HoleException` on line 137.

CodeTrek leverages (1) a declarative program analysis framework to produce a rich, easily extensible representation of context as a relational database, and (2) a biased graph-walk mechanism to prune that context in a task-specific way before presenting it to a model

based on Transformers (Vaswani et al., 2017) and Deep Sets (Zaheer et al., 2017). When implementing CodeTrek, I build it upon Semmle (Avgustinov et al., 2016), which converts codebases in C, Java, Python, etc., into relational databases that capture the underlying structure and semantics of code, as well as a query language, CodeQL, for specifying program analyses to compute new semantic information. Relational databases of code are not readily suitable for embedding into a continuous representation. I thereby introduce a novel and efficient procedure to convert a relational database into a graph.

Figure 3.2: CodeTrek's logo. This logo is a very close adaptation of StarTrek™ logo.

I evaluate CodeTrek on four diverse tasks on real-world Python programs. They include two existing tasks, *variable misuse* and *exception prediction*, as well as two newer ones, *unused definition* and *variable shadowing*. The newer tasks are sophisticated CodeQL queries, written by program analysis experts, and enable testing the power of neural models: they both involve complex logical reasoning, and only 1.6% of the *unused definition* samples contain bugs, which is more in line with real-world settings. CodeTrek achieves an accuracy of 91%, 63%, 98%, and 94% on these tasks respectively, which is 2-19% points higher than state-of-the-art neural models CuBERT, GREAT, GGNN, and Code2Seq. I also demonstrate the robustness of CodeTrek in two out-of-distribution scenarios: real-world *variable misuse* samples from GitHub and *unused definition* samples involving subtle code perturbations introduced using a systematic test-generation framework, Skeletal Program Enumeration (Zhang et al., 2017). CodeTrek achieves an accuracy of 57% and ROC-AUC of 78%, respectively in these scenarios, which is 6–11% points and 14–36% points higher than the baselines[1].

[1] CodeTrek is available at https://github.com/ppashakhanloo/CodeTrek.

Chapter Organization. I present an overview of CodeTrek in Section 3.1, elaborate on each component of CodeTrek in Section 3.2, and introduce a task specification language in Section 3.3. Then, I empirically and qualitatively evaluate CodeTrek in Section 3.4 and Section 3.5. Finally, I discuss limitations and improvements in Section 3.6.

3.1. Overview

Inspired by the idea of storing codebases as databases, CodeTrek represents a program as a relational database. Specifically, CodeTrek leverages the per-language schema defined by Semmle to uniformly store lexical, syntactic, and semantic program information as *base relations* in the database—I focus on Python in this chapter, but the approach is language-agnostic, as long as Semmle supports the language. Each relation contains information—in the form of tuples—about a particular kind of program element, such as expressions, statements, and so on. The columns of a relation specify its attributes. For instance, in Figure 3.1, tuple `(s1, except, c1)` in the `stmt` relation specifies that `s1` is an `except` statement contained in a scope with identifier `c1`, and tuple `(s4, o)` in the `def` relation specifies that variable `o` is defined in some statement with identifier `s4`. The schema also defines *referential integrity constraints* of the form `R.A` $\rightarrow$ `S.B` where `A` is called a foreign key of referencing relation `R`, and `B` is a unique attribute (e.g. a primary key) of referenced relation `S`. For example, in Figure 3.1, one such constraint is `stmt.CID` $\rightarrow$ `scope.ID`.

Facilitated by CodeTrek's uniform representation of programs, task developers can easily obtain new semantic information by writing program-analysis queries in CodeQL, an SQL-like language. The newly derived information is also in the form of *derived relations*, which maintains the uniformity of the relational representation. For example, in Figure 3.1, the derived information is stored in `def`, which, together with `call`, can bias the prediction of the most appropriate variable to replace a placeholder. A task developer need not be a machine-learning expert to bring in more semantic information about programs: All they need to do is write a CodeQL query, and the resulting derived information will be added to the existing richness of the program's available features in CodeTrek.

CodeTrek translates a relational database to a graph whose nodes correspond to tuples, and whose edges follow referential integrity constraints. An example of such a graph is illustrated in Figure 3.3 where each node is depicted as a circle along with its type (e.g., `func`) in white font. Orange and green nodes correspond to tuples of base and derived

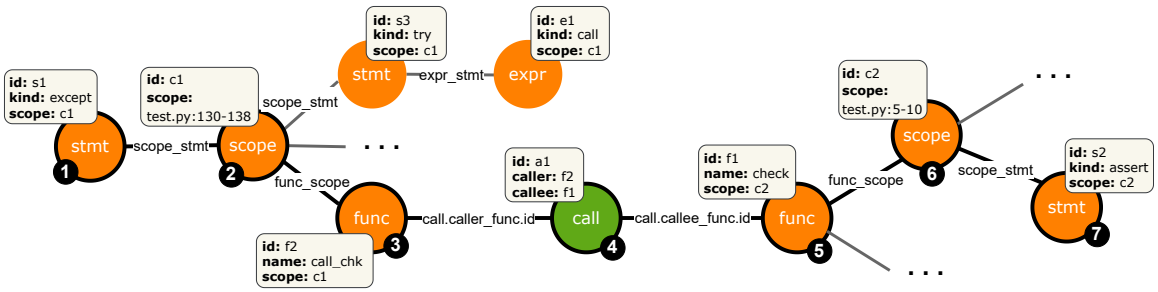

Figure 3.3: A partial illustration of a graph generated by CodeTrek.

relations, respectively. The attributes (e.g., `name`, `kind`, etc.) of the node are shown in a box at the corner of the node. For each referential integrity constraint `R.A` → `S.B`, an edge type `R.A_S.B` is defined, connecting the tuples of the two relations with the same value on the edge attributes `R.A` and `S.B`. For brevity of presentation in Figure 3.3, when there is a single such constraint between a pair of relations `R` and `S`, I omit the attributes from the edge type. But I do not omit them when there are multiple such constraints, such as in the case between relations `func` and `call`, namely, `call.caller` → `func.id` and `call.callee` → `func.id`. This graph view of program semantics helps extract succinct context as input to a model. Context extraction from the resulting CodeTrek graph is done via biased random walks of the graph, in a fashion specified by a walk specification. The starting node—which I call an *anchor* node—may be example-specific (e.g., the node containing the placeholder) or task-specific (e.g., all nodes holding a variable declaration). In Figure 3.3, the node that represents tuple `stmt(s1, except, c1)`—which corresponds to the statement on line 137 in Figure 3.1 (left)—is the anchor because the goal of the task is to predict a suitable exception type in the `except` statement. The walk generator traverses the graph by biasing traversal of edges according to each neighbor's node type. If no bias is specified, walks are simply fair random walks. Different probability mixes for different node types encourage the model to sample walks more relevant to a task. An example of such a walk is shown in Figure 3.3 using circles with thicker borders. This walk reaches the "`assert`" node which in fact determines the exception type that should be used in the except statement. For instance, to spend more time traversing longer-range dependencies in other

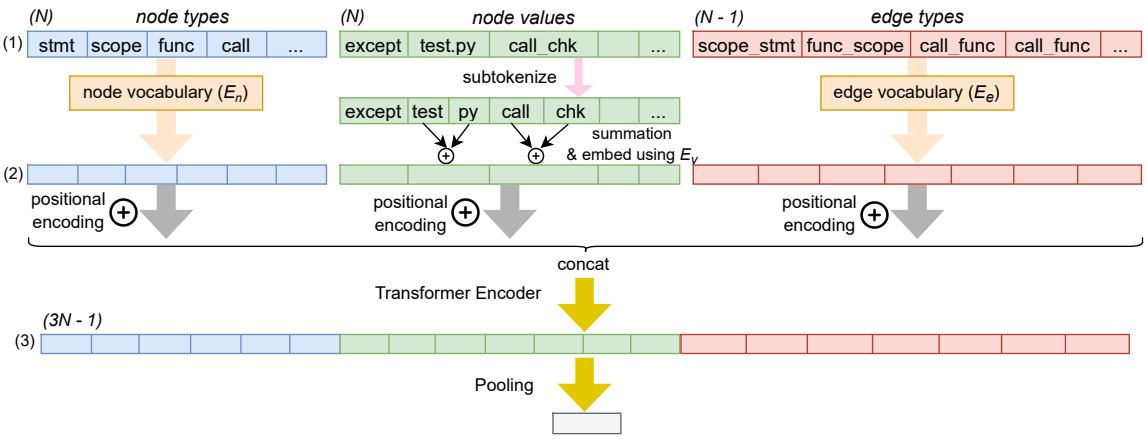

Figure 3.4: Embedding of the walk highlighted in Figure 3.3.

functions, the developer can assign a higher value to `call` nodes. For evaluation, I assign higher probabilities to nodes of types `stmt` and `expr`. I could achieve improved accuracy for the tasks compared to baselines by only modifying the probabilities of up to 4 types of nodes. Learning the probabilities in the walk specification is discussed in Chapter 4.

Finally, to convert random walks to a distributed representation, CodeTrek embeds each walk—including the types and attributes of each node and edge in the walk—using a Transformer encoder (Section 2.1.2), and then produces an order-invariant representation of the set of walks using the Deep Set architecture (Section 2.1.3). Using the Deep Set architecture rather than potential alternatives is driven by the following reasons: (1) CodeTrek needs to handle a different number of walks. The alternative approach of concatenating the walks would not allow that as it would change the size of the neural network. (2) The collected walks should be considered a *set*, not a *sequence*; so the order of the collected walks should not matter to the neural network. The resulting hidden representation can then be used by a decoder of choice to make predictions for the particular code-reasoning task.

Random walks to embed a graph are perhaps a regressive choice, compared to more modern solutions such as GNNs or relational Transformers. For instance, DeepWalk (Perozzi et al., 2014) uses random walks for distributed-representation learning in an online setting, and Code2Seq (Alon et al., 2018) uses shortest paths between pairs of AST leaf nodes. I chose

biased random walks for several reasons. First, this enables CodeTrek to choose task-specific strategies to heuristically fetch relevant context for a task, rather than choosing to embed all tokens of a function or class in lexical order. Second, in contrast to GNNs, this enables CodeTrek to potentially follow much longer chains through the semantic graph than what would be possible in a message-passing GNN of a tractable number of layers—this was, in fact, the motivation behind the model architectures in Hellendoorn et al. (2020). Finally, in contrast to other walk-based approaches such as Code2Seq (Alon et al., 2018) and AnyCodeGen (Alon et al., 2020), which share some of my motivations, my proposed graph structure has much richer connectivity and is larger. Code2Seq and AnyCodeGen only consider paths ending at token-bearing leaf nodes, with only AST interior nodes in between, whereas CodeTrek admits arbitrary paths through the graph. For instance, paths with more than two token nodes that are not at the end-points, e.g., the walk illustrated in Figure 3.3 using circles with thicker borders is admissible for CodeTrek; not for Code2Seq. CodeTrek builds upon the above techniques and extends to relational graphs, with different sampling strategies and neural architectures better suited for database representation.

3.2. Building Blocks

In this section, I describe the building blocks of CodeTrek. Preprocessing consists of two steps: (a) turning the codebase into a relational database (*Code2Rel*) using a pre-existing set of analyses and the developer's own analyses, and (b) mapping the relational database to a graph (*Rel2Graph*). Then, a graph is processed into sets of graph walks to present to a model. Finally, I train the models using a cross-entropy loss to implement a particular task. In the rest of this section, I describe the algorithms that CodeTrek uses for modeling source code.

3.2.1. Translating Source Code to Relational Database

Code2Rel applies to the codebase the system's base program analysis queries, which make up the base relations, as well as those provided by the developer, which form the derived relations. The result is a database comprising a number of named tuples for each relation

	Base Relations	Derived Relations
Python	95	277
Java	90	462
C++	182	553

Table 3.1: Universe size of the base and derived relations that are defined by Semmle.

type. For Python, I collected 95 base relations and 277 derived relations, although adding more derived relations is simply a matter of writing a few lines of CodeQL. Algorithm 1 formally describes how source code is transformed into tuples.

CodeTrek views program information as relations. Table 3.1 shows the tentative number of relations in three common programming languages. For example, $\mathbb{F}_{Python}$ represents the dependencies between 95 base relations plus 277 derived relations.

Note 1: Base and derived relations are both represented in the same way, so there is no fundamental difference between them. It was merely a choice to distinguish between them in the text so that it is easier to understand which relations are cheap to compute (i.e., base relations) and which ones are more costly (i.e., derived relations).

Note 2: In Algorithm 1, the topological ordering is independent of the program P but depends on the programming language in which P is written.

Algorithm 1 (**Code2Rel**) Given a program P, a set of base relation names R_B, and a set of derived relation names R_Q, construct and return database D. Note that the term *node* in this algorithm refers to nodes in the relation dependency graph, not to *nodes* in the program graph (e.g., in Algorithm 2) that the model will see.

---

1. Initialize D to the set of all base relations in R_B by translating program P.
2. Let R_S be the set of relation names reachable from R_Q in relation dependency graph $\mathbb{F}$:
   (a) $R_Q \subseteq R_S$
   (b) if $r \in R_S$ and $r \to r' \in \mathsf{Edges}(\mathbb{F})$ then $r' \in R_S$
3. Let F be the sub-graph of $\mathbb{F}$ induced by set of nodes R_S:
   (a) $\mathsf{Nodes}(F) = R_S$
   (b) $\mathsf{Edges}(F) = \{\ r \to r' \in \mathsf{Edges}(\mathbb{F}) \mid r, r' \in \mathsf{Nodes}(F)\ \}$
4. Compute a topological ordering $L = [r_1, ..., r_{|Nodes(F)|}]$ of F.
5. For each r in L in order:
   Evaluate the query for computing relation r on set D and add the result to D.

```
class TestObject:
  ...
  def aMethod(self):
    ...
    assert self.z == 0
...
class SyncTestCase:
  ...
  def testSync(self):
    o = TestObject()
    ...
    def callMethod():
      try:
        o.aMethod()
        ...
      except HoleException:
        errors.append(...)
```

Figure 3.5: Sample Python program.

3.2.2. Translating Relations to Program Graph

Rel2Graph interprets the relations produced by Code2Rel (Section 3.2.1) as a graph. In this graph, each named tuple is represented by a node with the values of the tuple attributes as its features. Edges are added between these nodes such that the edge type `R.A_S.B` is defined for each referential integrity constraint `R.A` $\rightarrow$ `S.B` between nodes representing tuples of relations `R` and `S`. Algorithm 2 formally describes how relations are translated into program graphs.

Algorithm 2 (**Rel2Graph**) Given a database D, construct a program graph G.

Construct an undirected and labeled graph G as follows:

(a) $\mathsf{Nodes}(G) = D$

(b) Add to $\mathsf{Edges}(G)$ each (t_1, t_2, l) that satisfies the following conditions:

i. $l : R.[a_1, ...a_k] \rightarrow S.[b_1, ..., b_k]$ is a referential integrity constraint in the schema of database D

ii. t_1 is a tuple of relation named R in D

iii. t_2 is a tuple of relation named S in D

iv. for all $i \in [1..k] : t_1.a_i = t_2.b_i$

Example 3.2.1. Algorithm 2 takes the relational database—i.e., the set of named tuples—that corresponds to the example program in Figure 3.5, and generates a program graph. In this graph, nodes are named tuples such as $\texttt{n}_1$`=func(f1,aMethod,s2)`, $\texttt{n}_3$`=call(f2,f1)`, and $\texttt{n}_2$`=func(f2,callMethod,s3)`. Tuples are connected to each other through referential integrity constraints. The type of the edge between two nodes is determined by a tuple of relation names and the common attributes, which is $\langle$`Functions.ID,Calls.Callee`$\rangle$ in this case.

3.2.3. Translating Graph to Walks

Given a code database that is converted to graph G via Rel2Graph (Section 3.2.2), I propose to represent it by the embedding of a set of walks W, via the procedure *Graph2Walks*. Graph2Walks projects a code graph as produced by Rel2Graph to a set of walks. These walks are generated according to a walk specification which defines the anchor node predicate, traversal bias, and target walk length. I will describe this specification language in more detail in Section 3.3. Graph2Walks samples from the distribution of such random walks, by repeatedly picking a node satisfying the anchor predicate, and traversing up to a maximum number of neighbors, following the transition probabilities specified.

Algorithm 3 (**Graph2Walks**) Given a program graph G, walk specification $S = \langle \mathcal{C}, B, min, max \rangle$, and the number of walks w, sample a set of walks W.

---

1. Initialize the set of walks $W = \emptyset$.
2. Compute the set of anchors $A = \{t | t \in \mathsf{Nodes}(G) \wedge t \text{ conforms to } S.\mathcal{C}\}$.
3. While $|W| \leq w$:
   (a) Pick a random tuple t_{curr} from A.
   (b) Construct $walk$ by repeating the following steps between $S.min$ and $S.max$ times:
      i. Set $t_{prev} := t_{curr}$.
      ii. Set t_{curr} to a $t \in \mathsf{Neighbors}(G, t_{prev})$ with prob. proportionate to $S.B[type(t)]$.
      iii. Let $e_{curr} = (t_{prev}, t_{curr}, l) \in \mathsf{Edges}(G)$
      iv. If $e_{curr} \notin walk$ then extend $walk$ by e_{curr}. Otherwise set $t_{curr} := t_{prev}$.
   (c) If $walk \notin W$ then add it to W.

The resulting walks are collected as token sequences of relation types of nodes, their attribute values, and the edge types traversed, in the order of traversal. The first row in Figure 3.4 shows such a walk representation corresponding to the walk highlighted in Figure 3.3. Al-

gorithm 3 formally describes how a set of walks is sampled from a program graph.

Example 3.2.2. The sequence of edges that are numbered ❶ through ❼ in Figure 3.3, is an example of a walk sampled from program graphs. The anchor of this walk is $\mathsf{stmt}(\mathsf{s1}, \mathsf{except}, \mathsf{c1})$, and the length of this walk is 7—the number of nodes along the walk.

3.2.4. Integrating all the Components

The algorithms described in Section 3.2.1, Section 3.2.2, and Section 3.2.3 are pieced together to generate walks from raw source code as shown in Algorithm 4.

Algorithm 4 (**Code2Walks**) Given a program P and a task specification $T = \langle R_B, R_Q, S, n \rangle$, generate a set of walks W.

---

1. $D = \mathsf{Code2Rel}(P, T.R_B, T.R_Q)$
2. $G = \mathsf{Rel2Graph}(D)$
3. $W = \mathsf{Graph2Walks}(G, T.S, T.n)$

3.2.5. Embedding Sampled Walks

Given a walk $w = [n_0, e_0, n_1, e_1, \ldots, n_{N-1}]$ of length N steps consisting of N nodes and $N-1$ edges, CodeTrek produces an initial embedding $X_w \in \mathbb{R}^{(3N-1)\times d}$, where d is the embedding dimension. It consists of three segments. The first N rows of the embedding tensor represent the N node types (relation names), using an embedding lookup in $E_n \in \mathbb{R}^{R\times d}$, where R is the number of relations. The next N rows represent the attribute values of the N nodes; I subtokenize attribute values (using a V-sized WordPiece vocabulary for attribute values), embed each subtoken using $E_v \in \mathbb{R}^{V\times d}$, and mean-pool the subtoken embeddings into each node's attribute embedding. The last segment represents the $N-1$ edge types (recall that an edge type is a tuple of two relation names and primary-key/foreign-key attributes), using an embedding lookup in $E_e \in \mathbb{R}^{I\times d}$, where I is the number of referential-integrity constraints in the database. All three embedding matrices E_n, E_v, E_e are learnable parameters. Each individual part of the embedding tensor gets its own sinusoidal positional encoding (Section 2.1.2) denoted as $\mathrm{P}_w \in \mathbb{R}^{(3N-1)\times d}$. I use a Transformer encoder to

represent the embedding of walk w as

$$\mathbf{e}_w = \textit{pooling}\Big(\text{Transformer}(X_w + \text{P}_w)\Big) : \mathbb{R}^{(3N-1)\times d} \mapsto \mathbb{R}^d, \tag{3.1}$$

where after the last layer of Transformer I perform mean-pooling over all $3N-1$ elements of the walk, to obtain a d-dimensional vector $\mathbf{e}_w$. The steps for embedding a walk sampled from the graph in Figure 3.3 are illustrated in Figure 3.4.

3.2.6. Training and Inference

An example consists of a set of walks and a ground-truth label, $(W, \hat{y})$. Given an unordered set of walk embeddings $\{\mathbf{e}_w\}_{w\in W}$, I build a classifier by using the construction

$$y = \text{MLP}\big(\text{DeepSet}(\{\mathbf{e}_w\}_{w\in W})\big), \tag{3.2}$$

where y denotes the predicted label, and optimize for cross entropy loss. For binary classification tasks, I use a more interpretable model via

$$y = \sum_{w\in W} \alpha_w \sigma(\text{MLP}(\mathbf{e}_w)), \tag{3.3}$$

where $\sigma(\cdot)$ is the sigmoid function, and

$$\alpha_w = \frac{\exp \text{MLP}(\mathbf{e}_w)}{\sum w' \in W \exp \text{MLP}(\mathbf{e}_{w'})}. \tag{3.4}$$

This way, one can inspect the individual walks that contributed the most (i.e., the highest α_w) to the positive or negative predictions, and see how that aligns with human reasoning. I refer to α_w as the *walk score*. I train the models using the Adam optimizer with 8 GPUs.

3.3. Task Specification Language

In this section, I present a specification language to define new and existing code-reasoning tasks. CodeTrek builds graphs and generates walks during task-specific training based on the specifications written in this language.

Task specifications describe base relation names, derived relation names, a number that specifies the maximum number of walks to generate when training a model for a specific code-understanding task, and a walk specification. A walk specification defines the length, anchors, and relation weights for a set of walks. More precise definitions of task specification and walk specification can be found in Definition 3.3.1 and Definition 3.3.2, respectively.

Definition 3.3.1 (Walk specification)**.** A walk specification $S = \langle \mathcal{C}, B, min, max \rangle$ is a tuple in which $\mathcal{C}$ is a conditional expression that filters walk anchors from the set of nodes in graph G, B is a map of bias values that correspond to each relation name, and $min, max \in \mathbb{N}^+$ specify the minimum and maximum length of the walks generated by the specification.

Definition 3.3.2 (Task Specification)**.** A task specification $T = \langle R_B, R_Q, S, n \rangle$ is a tuple in which R_B is a set of base relation names, R_Q is a set of derived relation names, S is a walk specification as described in Definition 3.3.1, and n is the number of walks to be generated.

3.3.1. Grammar of Task Specification Language

The syntax of the task specification language is shown in Table 3.2. For the full language specifications please refer to Appendix A. The semantics of *formula*, *predicate*, etc. follows the semantics that QL language reference[2] defines. The set of nodes (i.e., tuples) that satisfy the *predicate* are the anchors (i.e., the starting points) of the walks.

Note 1: Distinguishing between the base and the derived relation is not fundamental to CodeTrek. This distinction allows us to easily distinguish between cheap (i.e., base) and costly (i.e., derived) relations.

Note 2: Since the optimal number of walks for achieving high performance depends on tasks, the number of sampled walks (**N** in Table 3.2) is included in the task specification.

Note 3: The *target* of a task is specified using a conditional expression which is a part of the walk specification and filters the walk anchors from the set of all graph nodes (i.e., expression **C** in Table 3.2).

[2]https://codeql.github.com/docs/ql-language-reference/ql-language-specification/.

walk_spec ::=	**{**	*task_spec* ::=	**{**
C =	*predicate* **,**	**RB** =	*relations* **,**
B =	*scores* **,**	**RQ** =	*relations* **,**
min =	INT **,**	**S** =	*walk_spec* **,**
max =	INT	**N** =	INT
}		**}**	
scores ::=	**{** *score_tuples* **}**	*relations* ::=	**{** *rels* **}**
score_tuples ::=	*score* | *score_tuples*	*rels* ::=	*rel* | *rels*
score ::=	*rel* **:** INT **;**	*rel* ::=	relationName
predicate::=	**{ x :** *formula* **}** | **random**		
formula ::=	*fparen* | *disjunction* | *conjunction* | *negation*		

Table 3.2: Task Specification Syntax (Short Version). Tokens in **bold** font are language keywords and tokens in *italic* are non-terminals. The keyword **x** refers to any node, and the formula that comes after the colon (**:**) in front of it is used to filter specific nodes. INT is an integer literal. If no score is specified for a relation name, then its score is initialized to 1. Moreover, **random** is a keyword that can be used when no particular conditions are defined on anchors and they should be sampled randomly. "relationName" is any available relation name. For a full list of available Python relation names refer to Appendix B. For the full Task Specification Syntax refer to Appendix A.

3.3.2. Specifications for Evaluated Tasks

Using the grammar defined in Section 3.3.1, we describe the specifications of multiple tasks. In these example specifications, ellipsis (...) indicates the rest of the universe of base relations as designed in the Semmle framework. The specification language does not include it; it is used merely to simplify the text.

VarMisuse-Fun

```
{
 RB = { "stmt"; "var"; "expr"; "ssa-defn"; ... },
 RQ = { },
 S = {
   C = { x : (x instanceof "expr") and (x.kind is "name") },
   B = { "stmt": 5; "expr": 5; "var": 5; },
   min = 4,
   max = 16
 },
 N = 500
}
```

Exception-Fun

```
{
 RB = { "stmt"; "var"; "expr"; "ssa-defn"; ... },
 RQ = { } ,
 S = {
   C = { x : (x instanceof "stmt")
             and (x.kind is "except")
             and (x.type is "HoleException") },
   B = { "stmt": 5; "expr": 5; "var": 5; },
   min = 4,
   max = 16
 },
 N = 100
}
```

Exception

```
{
 RB = { "stmt"; "var"; "expr"; "ssa-defn"; ... },
 RQ = { "call" },
 S = {
   C = { x : (x instanceof "stmt")
             and (x.kind is "except")
             and (x.type is "HoleException") },
   B = { "stmt": 5; "expr": 5; "module": 0; },
   min = 10,
   max = 24
 },
 N = 100
}
```

DefUse-Fun

```
{
 RB = { "stmt"; "var"; "expr"; "ssa-defn"; ... },
 RQ = { },
 S = {
   C = { x : (x instanceof "expr")
             and (x.kind is "name")
             and ((x.context is "write")
                  or (x.context is "param")) },
   B = { "stmt": 5; "expr": 5; "var": 5; },
   min = 4,
   max = 16
 },
 N = 100
}
```

VarShadow

```
{
 RB = { "stmt"; "var"; "expr"; "ssa-defn"; ... },
 RQ = { "call" },
 S = {
   C = { x : (x instanceof "var") and x.isGlobal() },
   B = { "stmt": 5; "expr": 5; "var": 5; },
   min = 4,
   max = 16
 },
 N = 100
}
```

3.4. Evaluation

3.4.1. Experimental Setup

Task Selection

I consider two main criteria in selecting tasks. The first is *locality*, which is determined by whether reasoning within a function typically suffices, or whether inter-procedural reasoning is required. The second is whether the task can be stated as a logic problem that can be solved using declarative queries that rely on a set of base relations. For creating datasets for such tasks, I use Semmle CodeQL queries; for the tasks that cannot be solved using declarative queries, I rely on available datasets. Tasks are listed next.

1. VarMisuse. Given a function and a variable accessed in it, predict whether the variable is misused. I also consider a variation of this task, VarMisuse-Fun (Kanade et al., 2020), that takes only a function and predicts whether *all* variables are used correctly in the function. Note that neither variation can be solved using declarative queries: given a well-formed program, no logic query can deterministically decide that a variable is misused, since that decision depends on the intended semantics.

2. Exception. Given a module containing a masked exception type in an `except` clause, predict the most appropriate built-in exception type out of 20 choices. I also consider a

variation of this task, EXCEPTION-FUN (Kanade et al., 2020), that is similar to EXCEPTION but takes a single function. Although EXCEPTION needs inter-procedural reasoning, neither variation can be solved using declarative queries, since the choice of the most appropriate exception type is subjective in Python.

3. DEFUSE. Given a function and a local variable definition, predict whether the definition is used. I also consider a variation of this task, DEFUSE-FUN, that takes a function as its input and predicts whether *any* definitions are unused. This task is especially interesting because the real-world distribution of programs that contain unused definitions is skewed. What makes this choice even more important is that "...there is a fundamental mismatch between the real bug distribution found in public code repositories and the synthetic bug distribution used to train and evaluate existing detectors..." and "...correct programs outnumber buggy ones in practice..." (He et al., 2022). So the results one obtains from this new dataset is potentially closer to what engineers encounter when using neural models in real-world scenarios. Both variations can be computed using logic queries and require only intra-procedural reasoning.

4. VARSHADOW. Given a module, predict whether any variable defined within a certain scope has the same name as a variable defined in an enclosing outer scope, thereby shadowing that latter variable. To understand the importance of this task, one needs to know more about scopes in Python. Python statements can access variables in both the local and global scopes. A local variable which has the same name as a global variable *shadows* or *hides* the global variable. Unless the developer explicitly references a global variable with the `global`[3] statement, the local variable is used. It can lead to confusion since a programmer may think the variable refers to a global[4][5]. Similar to EXCEPTION, the VARSHADOW task requires inter-procedural analysis to reason over both local as well

[3]https://docs.python.org/3/reference/simple_stmts.html#the-global-statement

[4]https://github.com/github/codeql/blob/main/python/ql/src/Variables/ShadowGlobal.qhelp

[5]One might wonder why I chose the task of detecting global shadowing over local shadowing. In Python, local shadowing is generally considered less important because it only affects the code within a specific scope and does not propagate to other scopes.

as global variables. Also, this task can be computed using a logic query.

Benchmark Suite

I use the ETH Py150 Open corpus consisting of 125K Python modules[6]. It is a de-duplicated and redistributable subset of ETH Py150[7]. Specifically, for the non-declarative tasks, I use the datasets released by Kanade et al. (2020). Since these are function-level samples but the Exception task is module-level, I augment the function in each sample with the entire containing module for this task. For the declarative tasks, I use the analyses written in CodeQL to annotate all functions (or modules, as applicable) in ETH Py150 Open. The analysis queries can be found in Appendix C. All datasets consist of real examples, except for VarMisuse-Fun and VarMisuse where variable misuses are synthetically introduced into real code. I use a number of apparent variable misuses from GitHub commits to test the models and baselines on a realistic dataset.

Baselines

To compare CodeTrek's performance with state-of-the-art techniques, I select four baselines: I use GGNN by Allamanis et al. (2018) and Code2Seq by Alon et al. (2018), build classifiers on top of the GREAT encoder by Hellendoorn et al. (2020), and fine-tune the pre-trained Python model for CuBERT by Kanade et al. (2020), which is essentially the Transformer-based classifier implementation of BERT. For Code2Seq, I use ASTs as base program structures as described by Alon et al. (2018). I sample leaf-to-leaf paths from these ASTs. The number of paths I sample is the same as the number of walks I sample for training CodeTrek models. For GGNN and GREAT, I compute the dataflow, control flow, and lexical information described by Allamanis et al. (2018) and Hellendoorn et al. (2020), respectively, using Semmle CodeQL and augment program ASTs with those edges.

Hyperparameters

The following section discusses the parameters and hyperparameters I used for training CodeTrek and the baselines.

[6]https://github.com/google-research-datasets/eth_py150_open
[7]https://www.sri.inf.ethz.ch/py150

Task	CodeTrek	GGNN	Code2Seq	GREAT	CuBERT
VarMisuse	**0.91** ± 0.003	0.72 ± 0.004	–	0.84 ± 0.002	0.89 ± 0.003
VarMisuse-Fun	0.70 ± 0.004	0.58 ± 0.004	0.52 ± 0.005	**0.86** ± 0.003	0.84 ± 0.003
Exception	**0.63** ± 0.003	0.30 ± 0.02	0.30 ± 0.01	0.45 ± 0.008	0.42 ± 0.008
Exception-Fun	0.65 ± 0.01	0.53 ± 0.02	0.51 ± 0.008	0.68 ± 0.007	**0.69** ± 0.007
DefUse ∗	**0.98** ± 0.002	0.78 ± 0.07	–	0.87 ± 0.05	0.76 ± 0.01
DefUse-Fun ∗	**0.91** ± 0.005	0.77 ± 0.07	0.66 ± 0.01	0.84 ± 0.007	0.71 ± 0.01
VarShadow	**0.94** ± 0.007	0.74 ± 0.01	0.70 ± 0.01	**0.94 ± 0.008**	0.91 ± 0.008

Table 3.3: Accuracy results of CodeTrek. Rows that are marked by ∗ are measured by ROC-AUC, and the rest are measured by accuracy. The best performance in each row is denoted in boldface.

CodeTrek I train CodeTrek models with a learning rate of 10^{-4}, 4 transformer encoder layers, an embedding size of 256, 8 attention heads, and 512 hidden units. I sample 100 walks with lengths of up to 24 in each graph for every task, except for the VarMisuse-Fun task for which I sample 500 such walks per graph. The reason is that the anchors I select for VarMisuse-Fun task are all the variables in the given program which can be well over 100 variables. So, I increase the total number of walks to include more random walks starting from each variable.

CuBERT I fine-tune the CuBERT pre-trained model that is provided by Kanade et al. (2020) with a learning rate of 10^{-4}, 4 transformer encoder layers, and 512 hidden units. I use the checkpoint[8] that is pre-trained on examples of size 512 tokens.

GREAT I train GREAT models with a learning rate of 10^{-4}, 10 transformer layers, 8 attention heads and 512 hidden units. The example size in 512 tokens.

Code2Seq I train Code2Seq models with a learning rate of 10^{-3}, 4 layers, 512 hidden units, and embedding size of 256. I sample 100 paths in each AST.

GGNN I train GGNN models with a learning rate of 10^{-4}, 10 layers, a latent dimension of size 128, and a message dimension of size 128.

```
1 def get_month(self, t):
2   month, _, _ = t
3   def validate(month):
4     return is_valid(month)
5   return self.month
```

Figure 3.6: Example DefUse-Fun task.

3.4.2. Accuracy

I evaluate the performance of CodeTrek and the baseline techniques on all the tasks described in Section 3.4.1, all presented as classification tasks. I report the average of the metric that I use to measure the performance of each task. I use ROC-AUC as the metric for DefUse and DefUse-Fun tasks due to their unbalanced datasets, and accuracy for the remaining tasks. In theory, ROC-AUC could be used in all cases. The reason I chose accuracy over ROC-AUC unless absolutely necessary is that accuracy can be computed, understood, and interpreted more easily compared to ROC-AUC. The results are reported in Table 3.3. In 5 out of 7 tasks, CodeTrek outperforms GGNN, Code2Seq, GREAT, and CuBERT by 2–19% points.

There are various reasons why CodeTrek performs better than these approaches. First, declarative tasks such as DefUse-Fun (or DefUse) require complex reasoning about the interactions between program variables. For instance, one needs to reason about the uses and definitions of variables in a flow-sensitive manner to determine whether any unused definitions exist in a program. Consider the code snippet in Figure 3.6. The definition of the variable `month` on line 2 is unused but that at line 3 is used in line 4. CodeTrek gives a majority of the walks sampled using the definition at line 3 a high score (around 0.99 out of 1), indicating the existence of a use of that definition. However, most of the walks sampled from the definition of `month` at line 2 were given a lower score, and so CodeTrek determines that this definition is unused. I observe that both CuBERT and GREAT fail to distinguish between the definitions on lines 2 and 3, and so they predict both to be used.

[8] gs://cubert/20210711_Python/pre_trained_model_epochs_2__length_512

Additionally, some tasks such as Exception require reasoning beyond the boundaries of a single function to make informed predictions. Functions in the chain of function calls can be lexically far from each other, thus rendering the limited context size of CuBERT and GREAT insufficient. I observe that GGNN fails in the presence of such long call chains on par with findings of Alon and Yahav (2020). CodeTrek addresses this kind of mispredictions by readily using a call graph relation to connect the chain of function calls. This enables CodeTrek to traverse a long distance without the need to consider other statements in the program that have no effect in raising an exception.

However, CodeTrek performs worse than CuBERT in Exception-Fun. This could be attributed to the fact that CuBERT is pre-trained on around 7 million Python programs, and therefore is able to memorize tokens from several instances of `try-except` blocks. An example of a heuristic that it learns is that in presence of tokens such as `request` or `response` in the context, it suggests catching `HTTPError`, which is usually the correct choice. However, its prediction is not robust against changes in the variable names. For instance, changing the names of a few nearby variables to `request` or `response` forces CuBERT to predict `HTTPError` regardless of the semantics. CodeTrek on the other hand, does not rely on memorizing the tokens, but learns to assign high probabilities to walks that correctly traverse a chain of function calls starting from the `try` blocks to locations in programs (or their libraries) where the exception is originally raised.

CodeTrek also performs worse than GREAT in VarMisuse-Fun. This is because every node that corresponds to a variable is selected to be an anchor for this task. The total number of walks (500 in this task) is divided among these variables. However, there can be hundreds of variables in some programs, resulting in few walks generated for each variable in such cases, diminishing the ability of CodeTrek to learn sufficient information about each variable.

In summary, CodeTrek outperforms state-of-the-art in complex code-reasoning tasks that cannot merely rely on memorizing source code tokens.

Task	CodeTrek	GGNN	Code2Seq	GREAT	CuBERT
VarMisuse-Real	**0.57**	0.51	0.50	0.49	0.46
DefUse-SPE	**0.78**	0.53	0.63	0.41	0.47

Table 3.4: Robustness results of CodeTrek.

3.4.3. Robustness

I evaluate the robustness of CodeTrek on additional test data that does not follow the distribution of the training data. This data includes two new datasets: one representing real-world bugs for the VarMisuse-Fun task and the other consisting of programs mutated using a systematic test-generation framework for the DefUse-Fun task. The results are reported in Table 3.4.

Real-world bugs. I manually collect 199 real-world instances containing a VarMisuse-Fun bug and their corrected counterparts (a total of 398 samples) from commits on GitHub and use them as testing data for the VarMisuse-Fun task. I define a VarMisuse-Fun bug as the occurrence of a misused variable that is changed to another in-scope variable in the commit. This dataset provides a set of examples that reflect the types of errors that can occur in actual software development. I evaluate the baselines using this real-world set of bugs. CodeTrek outperforms baselines in detecting real-world variable misuse bugs (VarMisuse-Real) by achieving an accuracy of 57% which is 6% points better than the second best result obtained by GGNN.

Mutated programs. There are several approaches to mutating existing datasets, including transforming existing data (Yang et al., 1992), generating synthetic programs, and fuzzing. A representative approach that has been used to systematically evaluate the robustness of compilers is Skeletal Program Enumeration (SPE), proposed by Zhang et al. (2017). SPE parameterizes each program by a set of its variables, and replaces each variable name exhaustively with other in-scope variable names. I generate variations of the DefUse-Fun testing data using this technique, and evaluate the baselines on this mutated dataset (DefUse-SPE). CodeTrek outperforms all baselines in classifying these perturbed pro-

grams by achieving the ROC-AUC score of 78% which is 15% points better than the second best result obtained by Code2Seq.

The poor performance of the baselines can be explained by the the fact that the code generated by SPE is out-of-distribution. For example, the assignment `a = a + a` is unusual in real code, but occurs frequently in SPE-generated samples. Despite this, the inductive bias borne by rich relational information during training remains applicable and prevails over the unusual-looking token sequences, thus explaining CodeTrek's performance.

These results suggest that sampling walks can be a promising strategy for robustness. Interestingly, the runner-up in this study is Code2Seq—another walk-based approach. I inspected the paths in both approaches to understand the reason behind the difference in performance of Code2Seq and CodeTrek despite their similarities. I identified two possible reasons: 1) the kinds of program information that can be captured from an AST are limited compared to the program graph I propose, and 2) several walks that CodeTrek prioritizes for this task cannot be embedded by Code2Seq.

3.4.4. Effectiveness on Long-range Tasks

I evaluate the effectiveness of CodeTrek on tasks that require reasoning beyond function boundaries. CodeTrek achieves this ability by readily incorporating relations that capture inter-procedural or inter-modular dependencies such as call graphs. To demonstrate this, I compare CodeTrek's performance on Exception with vs. without incorporating the call graph information at training time. CodeTrek achieves an accuracy of 52% when call graph information is not provided, which increases to 63% after providing the call graph information between functions within a module.

Consider the representative example in Figure 3.7, snipped and simplified from the `zipfile` package[9], to illustrate the types of mistakes that baselines (and also CodeTrek without call graph information) make. In this example, the model predicts the exception type that should be caught on line 11. However, to make an informed prediction, the model must

[9]https://github.com/python/cpython/blob/3.9/Lib/zipfile.py

```
1  class ZipFile:
2    def __init__(...):
3      self.__check_compression(...)
4    def __check_compression(...):
5      raise NotImplementedError
6  # ...2000 lines of code...
7  class TestZipFile:
8    def test(path):
9      try:
10       zf = ZipFile(path)
11     except HoleException:
12       log.warning()
```

Figure 3.7: Example EXCEPTION task.

consider the exceptions that may be raised when calling the `ZipFile` constructor (line 10). Hence, the definition of the constructor (line 2) must be taken into consideration. This constructor calls another function `__check_compression` in which a `NotImplementedError` is raised on line 5. This chain of function dependence can be easily represented using a call graph. Therefore, CODETREK, once provided with a call graph, will eventually traverse the path that reaches this `raise` statement from the `except` statement through call graph edges.

3.4.5. Sensitivity Analysis

Number of Walks

I evaluate the sensitivity of CODETREK at test time to the number of walks that are sampled from program graphs. Almost all the models for the considered tasks are trained on 100 sampled walks per program. Only VARMISUSE-FUN is trained on 500 walks due to the large number of anchor points this task requires. The results are reported in Figure 3.8. I observe that the accuracies of the models increase with the number of sampled walks. In some tasks, such as DEFUSE and VARMISUSE that involve local reasoning about one point in the program, reducing the number of walks from 100 to 50 reduces the accuracies of the models by a very small amount. On the other hand, for tasks that require reasoning about numerous points in the program (e.g., DEFUSE-FUN) or reasoning globally (e.g., EXCEPTION) decreasing the number of sampled walks has a bigger impact on the accuracy.

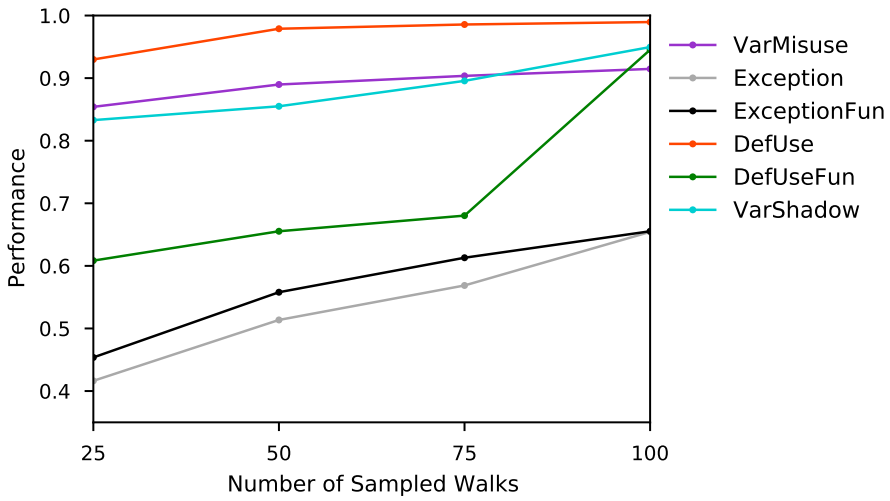

Figure 3.8: Sensitivity to the number of walks.

# **Steps**	4	6	12	18	24	30
Accuracy	0.41	0.49	0.60	0.63	0.64	0.65

Table 3.5: Sensitivity to the length of walks.

For instance, reducing the number of walks for the DefUse-Fun task from 100 to 75 reduces the accuracy of that model by around 26%.

Length of Walks

To measure the sensitivity of CodeTrek to the length (i.e., number of steps) of walks, I train a number of models for the Exception task with walks of length 4–30 steps. I report accuracy changes in Table 3.5. Longer walks tend to improve accuracy. Walks that are too short (4 or 6 hops) result in models with low accuracy (42% and 51%, respectively) because they are not able to capture enough information to make predictions. There is, however, a point when enough context is captured (e.g., 24 hop walks) and longer walks do not improve performance significantly.

3.4.6. Representation Impact

To evaluate the impact of different code representations, I train two models for each task using CodeTrek's architecture: for one set of models, the walks are sampled from rela-

Task	Relational	AST
VarMisuse	0.91	0.63
VarMisuse-Fun	0.70	0.55
Exception	0.63	0.37
Exception-Fun	0.65	0.62
DefUse	0.98	0.63
DefUse-Fun	0.91	0.67
VarShadow	0.94	0.73

Table 3.6: The impact of intermediate code representation on performance.

tional graphs whereas for the other set of models, the walks are sampled from ASTs. The performance results are reported in Table 3.6. Notably, the models trained on walks sampled from relational graphs are about 3–35% points more accurate than models trained on walks sampled from ASTs.

3.4.7. Importance of Bias in Random Walks

I evaluate the usefulness of the ability to bias random walks in CodeTrek. The accuracy results that are reported in Table 3.3 are all trained on biased random walks.

Specifically, in all of the tasks, nodes with types `stmt`, `expr`, and `variable` are biased such that they are 5 times more likely to be traversed compared to other kinds of neighboring nodes. In addition, in the Exception task, I decrease the bias assigned to nodes of type `module` to 0 to avoid traveling from one function to another through the module node they have in common. This forces the walks to only go to other functions by taking call graph edges between them. I select one of the tasks, Exception-Fun, to measure the accuracy in the absence of said biases. I re-train Exception-Fun using uniformly sampled walks and observe that the accuracy reduces from 65% to 58% as a result.

3.4.8. Ablations

I measure the contribution of the positional encoding which is used in embedding walks, the impact of derived relations in improving the accuracy, and the effect of the biases assigned to node types. I report the results in Table 3.7. Every row in this table shows a different configuration indicated by 1-4.

Config #	Positional Encoding	Biases	Derived Relations	Accuracy (%)
1	✓	✓	✓	63.83
2	×	✓	✓	62.06
3	×	×	✓	55.76
4	×	×	×	45.19

Table 3.7: Contribution of different factors to accuracy of EXCEPTION task.

I train a model for the EXCEPTION task while using the positional encoding in embedding the components of each walk, the `call` relation that shows the relationships between functions and their callers, and the biases which are assigned to nodes of type `stmt`, `expr`, and `variable` (Config 1). This setting is similar to that of Table 3.3 in Section 3.4.2. With this setting, CODETREK achieves an accuracy of 63.83% on the EXCEPTION task. If I remove the Positional Encoding (Config 2), I note a small drop of 1.77% points in the accuracy. The effect of further removing the biases (Config 3) is much higher: CODETREK 's accuracy drops 6.3% points from 62.06% to 55.76% points. This aligns with the intuition that adding biases to the aforementioned node types results in generating walks that are more relevant to the task. Finally, I obtain the largest drop in the accuracy by further removing the derived `call` relation (Config 4). This component contributes a significant amount of 10.57% points to the accuracy of the task, and it obtains a low accuracy of 45.19% points in absence of all three components.

Different Pooling Mechanisms

I examine the effect of mean pooling versus attention pooling on the performance of CODETREK models. Attention pooling increases the accuracy of the EXCEPTION task from 63.83% to 66.43%.

Different Positional Encoding Techniques

I also explore different positional encoding alternatives. The current setting of CODETREK gives an accuracy of 63.83% in the EXCEPTION task. Substituting the sinusoidal positional encoding with a learned one improves the accuracy less than 1% points. This result is in line with findings of Vaswani et al. (2017), which report nearly identical results using both

```
def write_random_to_file():
    no = random.randint(1, 10)
    with open("random.txt", "w") as file:
        file.write(str(no))
    return no

def write_random():
    random_no = write_random_to_file()
    print "A random number was written to random.txt"
```

Figure 3.9: A simple code snippet for DefUse.

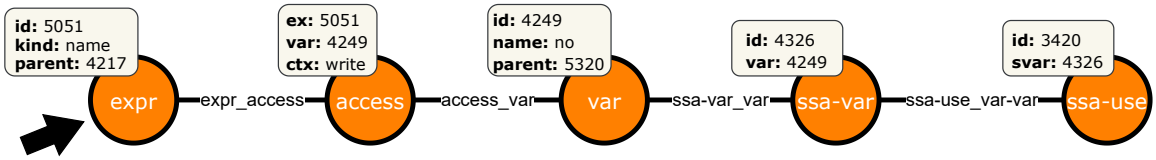

Figure 3.10: The most important walk in a simple instance of DefUse.

positional encoding techniques.

3.5. Qualitative Study

I discuss a few examples in this section. My focus is on describing walks that contribute most to predictions that CodeTrek models make. It will clarify how the relations and the semantic edges between them contribute to accurate predictions in CodeTrek. These examples showcase CodeTrek's intrinsic interpretability which is defined as the ability to explain or to present in understandable terms to a human (Hall et al., 2017). In Chapter 5, and more specifically in Section 5.4, I discuss CodeTrek's interpretability in greater detail.

3.5.1. Example: Detecting Unused Definitions

To understand how CodeTrek uses semantic relations to determine whether a defined

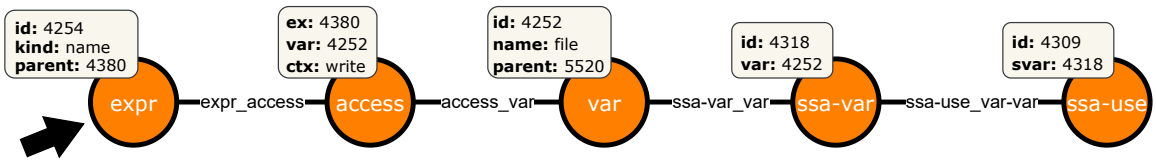

Figure 3.11: The most important walk in a challenging instance of DefUse.

```
1 def construct_file_handle():
2     file = Handler.initialize()
3     def check_handle(file):
4         if file.id < MIN_H:
5             return False
6         return True
7     file = Handler.initialize()
8     return Handler.default()
```

Figure 3.12: A challenging code snippet for DefUse.

variable is used, consider the code snippet listed in Figure 3.9. In this snippet, the local variable `file` is defined on line 3, and then accessed on line 4. A programmer would start with the variable definition and then follow the code to find an access to it to prove that it is indeed a used variable. More specifically, a programmer starts with the expression on line 3 in which the variable `file` is declared. She then tries to find another access to this variable that reads it, such as on line 4.

CodeTrek determines that the walk illustrated in Figure 3.10 has the highest score among a set of randomly generated walks. Interestingly, this walk shows a similar behavior to that of a programmer: it starts at the anchor node (an `expr` node corresponding to the variable definition) which corresponds to the expression that defines `file`. Then, it traverses the graph towards a node that corresponds to a use of this variable (a `ssa-use` node corresponding to the variable use).

Even the models that do not embed semantic edges (e.g., CuBERT) are able to correctly predict that `file` is used, in such simple cases. However, in more complicated cases, such as the code snippet listed in Figure 3.12, semantic edges are needed to be able to distinguish between different definitions of variable `file` and to not confuse various uses of them. In this snippet, `file` is defined on line 2, and then on line 7. The `file` defined on lines 2 and 7 are never used. To make matters more complicated, there is a function `check_handle` that is defined inside the top-level function `construct_file_handle`. This function takes an argument which is named `file` and uses it on line 4.

In the absence of edges that make the relationship between uses and definitions of variable explicit, it is challenging for a model to determine that the variable named `file` on line 2 is different from the variable of the same name on line 4. As a result, GREAT and CuBERT fail to label the variable definition on line 2 correctly. CodeTrek, however, takes advantage of the relationship between the variable definition and its use (an `ssa-use` node) and makes a robust prediction. Among the set of randomly generated walks starting from the definition on line 2 (`expr` node with `id` 4244) there are no walks with the following pattern which only occurs when a variable is used after being defined: "`expr` → `access` → `var` → `ssa-var` → `ssa-use`". Therefore, CodeTrek predicts that this variable is never used. On the other hand, as illustrated in Figure 3.11, a walk with the mentioned pattern exists between the definition on line 3 (`expr` node with `id` 4254) to its use on line 4 (`ssa-use` node with `id` 4309). So, CodeTrek predicts that this variable is used. It is worth emphasizing that the walks illustrated in Figure 3.10 and Figure 3.11 are very similar although they correspond to completely different code snippets.

3.5.2. Example: Predicting Appropriate Exceptions

For the Exception task, I choose a code snippet from the test dataset, which is listed in Figure 3.13. In this code snippet, CodeTrek must predict the exception to be caught by the `except` statement at line 34 (represented by `HoleException`). The correct exception type is `ValidationError`. Due to the fact that `admit_car` is called within the corresponding try block, I inspect the body of `admit_car` and notice that it may raise `ValidationError`.

Out of the walks sampled by CodeTrek for predicting the correct exception, I illustrate the most important (highest scoring) walk in Figure 3.14. This walk represents the aforementioned intuitive reasoning for predicting the exception. It starts at the anchor node, which is the node of type `stmt` with `id` 4506, corresponding to the `except` statement on line 34. It traverses to the function definition of `admit_car` by first traversing to the `call` node for `admit_car` with `id` 5253, representing the call on line 32, and then following the corresponding call graph edge to the definition of `admit_car`. These call graph edges al-

```
class Admission:
  ...
  def admit_car(self, car):
    if not str(car.get_id()).isdecimal():
      raise ValidationError("Index is not valid")
    name = car.get_number()
    if name.upper() != name:
      raise ValidationError("Number not in capslock")
    if name.count(" ") < 2:
      raise ValidationError("Number needs 3 parts")
    check_name = name.split(" ")
    if not check_name[0].isalpha():
      raise ValidationError("First part is not alpha")
    if not check_name[1].isdecimal():
      raise ValidationError("Second part not decimal")
    if not check_name[2].isalpha():
      raise ValidationError("Third part is not alpha")
    owner = car.get_owner().replace("-", " ")
    if not owner.isalpha() or not owner.istitle():
      raise ValidationError("Owner's name not correct")
    if len(owner) > 40:
      raise ValidationError("Name too long")

# a number of unit test functions removed here
# only for presentation purposes ...

def test_car_admit():
  admit = Admission.get_instance()
  car1 = Car(1, "ag 12 BOB", "Dan")
  car4 = Car(4, "A", "Ian")
  try:
    admit.admit_car(car1)
    assert False
  except HoleException:
    assert True
```

Figure 3.13: A sample code snippet for Exception.

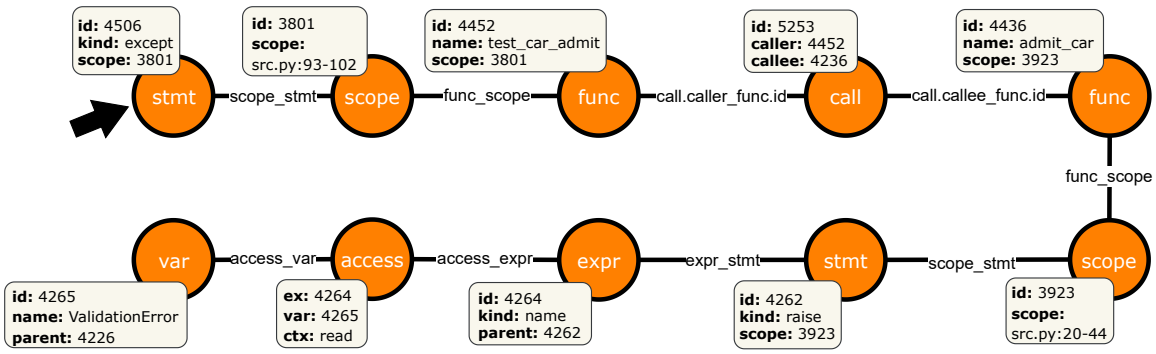

Figure 3.14: The most important walk in an instance of EXCEPTION.

low for such inter-procedural reasoning. The walk then traverses to the `stmt` node for the `raise` statement, then to its expression, and reaches the `ValidationError` exception via its corresponding `access` node.

3.5.3. Example: Detecting Shadowed Global Variables

VARSHADOW is an example of a long-range task in which the model is expected to distinguish between the global and the local scopes in order to predict whether a global variable is shadowed by another variable with the same name that is defined in a local scope. I use the code snippet in Figure 3.15 to explain how semantic relations help in such tasks.

To determine whether a global variable is shadowed by a local variable, a programmer would look for variables that are defined in local scopes and have the same name as the global variable. The walk which is illustrated in Figure 3.16 captures the relationship between the global variable definition (`expr` node with `id` 5051) and a local re-definition with the same name (`access` node with `id` 4346) by visiting a local scope (`scope` node with `id` 8456) of the module (the `module` node) along the way. Interestingly, CODETREK assigns the highest importance to this walk among a number of randomly generated walks, and can therefore correctly predict that `env_vars` is a shadowed global variable.

```
1  env_vars = env.vars
2
3  class SystemReq:
4    ...
5  ...
6  class Utils:
7    @staticmethod
8    def rev(s):
9      for i in range(len(s) // 2):
10       tmp = s[i]
11       s[i] = s[-(i+1)]
12       s[-(i+1)] = tmp
13
14   @staticmethod
15   def env_check():
16     env_vars = environ.vars
17     return env_vars
```

Figure 3.15: A sample code snippet for VarShadow.

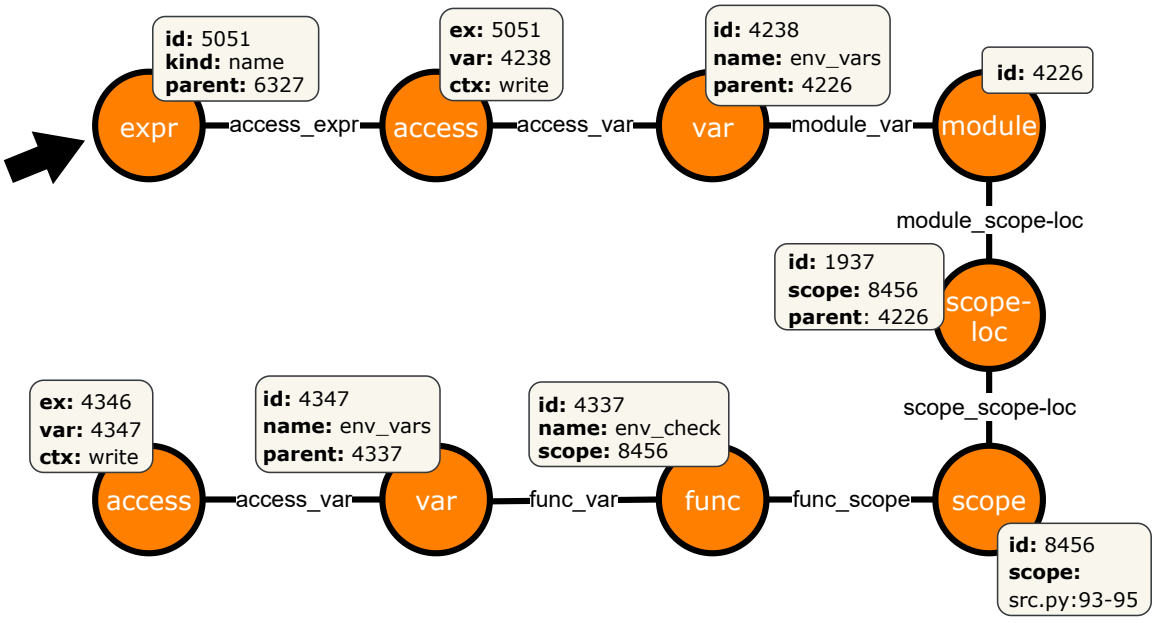

Figure 3.16: The most important walk in an instance of VarShadow.

3.6. Discussion

Despite its effectiveness, CodeTrek has some limitations which I discuss next.

Human Intervention. Writing the task specifications is manual in CodeTrek. The most difficult part is assigning weights to different relations. In Chapter 4, I introduce *walk policies* and present a technique to learn these weights.

Evaluated Architectures. CodeTrek's walk embedding uses a standard Transformer. Expanding to more recent variants or a different neural architecture might better serve CodeTrek. Also, I did not delve into other machine learning architectures for embedding the proposed relational graphs or walks.

Evaluated Tasks. I have only explored *binary and multi-class classification tasks* in this chapter. Later in Chapter 5, I implement a code-to-text application for evaluating the robustness and interpretability of the generated text when the representation learning is done via CodeTrek.

Pretraining. Another area for improvement is to pretrain models using objectives targeting all CodeTrek relational features, perhaps as an expansion of an approach such as GraphCodeBERT (Guo et al., 2020a), to explore the power of CodeTrek towards unsupervised transfer learning.

Computed Tasks. Even though CodeTrek is able to generate new labelled datasets for a large set of code-reasoning tasks, it is limited to labelling those that can be described via logic. Generating datasets for non-declarative tasks is an exciting direction but is beyond the scope of this dissertation.

Importance of Deep Sets. I explained in Section 3.1 that using Deep Set as a part of CodeTrek offers many advantages. It would be interesting to see whether it is considered indispensable in a future study.

Adversarial Experiments. I have only compared CodeTrek's robustness with a number of widely-known state-of-the-art models. However, a fairer and more interesting future

exploration is comparing CodeTrek with models that are specifically designed to withstand robustness-related attacks.

Coverage. Generally, the coverage of the graph increases as the number of sampled random walks and the walk length increase. However, exploring the correlation between the number of walks and code coverage is a topic for future work. A further open question concerns whether more walks improve the model's performance if they do not change the coverage.

3.7. Summary

In this chapter, I presented CodeTrek, a technique that represents programs as relational databases to make rich semantic information available to deep learning models for code-reasoning tasks. I also introduced a flexible walk-based mechanism to sample relevant contexts from large graphs which are constructed from relational databases. Finally, I evaluated CodeTrek on a variety of real-world tasks and datasets, and showed that it outperforms state-of-the-art neural models.

CHAPTER 4

Learning to Walk over Relational Graphs

Program representation is crucial to the effectiveness of deep learning techniques for program understanding. It is essential to understand the structural and semantic information contained in source code in order to represent programs effectively (Allamanis et al., 2018; Hellendoorn et al., 2020; Pashakhanloo et al., 2022b). While the abundance of information greatly benefits program representation techniques, it also poses a fundamental challenge. It is difficult to determine which kinds of program information are most relevant for a given task. There have been efforts in crafting features for programs. These approaches, however, require a lot of engineering and domain knowledge. AST-based (Alon et al., 2018, 2020) or control flow-based models (Hellendoorn et al., 2020; Allamanis et al., 2018) require domain experts to select what information to include and how to incorporate it.

CodeTrek (Chapter 3), alleviates this issue by modeling programs as relational graphs that uniformly store all available structural and semantic information. CodeTrek outperforms the state-of-the-art by representing programs as biased walks over rich relational graphs, thereby selecting some of the available information via the walk policy. This technique, however, does not provide an automated approach to discovering bias; a domain expert must determine which kinds of information (e.g., statements, successors) are more relevant for a given task. As I pointed out in Section 3.6, CodeTrek is limited by its reliance on domain knowledge for tackling new tasks. Also, using uniform random walks (Perozzi et al., 2014) does not help because, given a limited budget, as the space of possible random walks increases, the selection of suitable random walks for a given task becomes more difficult.

In this chapter, I present a deep learning approach to learn walk policies (Pashakhanloo et al., 2022a) to address challenges that stem from the abundance of available program information in relational graphs. The proposed policy-learning mechanism selects relevant relations in a task-specific manner by sampling a set of graph walks over the relations. It uses the

properties of relational graphs (relation names, tuples, and key-foreign-key relationships) to learn policies that guide the random-walk generation. In particular, it aggregates the attention weights that are computed while the model learns to embed each walk, to rank relations by their relevance to the task. Higher-ranking relations have a higher relevance, and are more likely to be visited during the guided random traversal. The mapping between relation names and their scores defines a *walk policy*.

I evaluate the proposed walk policy learning mechanism on various program-understanding tasks, and observe that the policies learned in this manner yield superior accuracy to even those hand-picked by a human expert[1].

Chapter Organization. I introduce and define the walk policy in Section 4.1, and elaborate on the training process in Section 4.2. Then, I evaluate the effectiveness of walk learning in Section 4.3. I discuss the interpretability of learned walks in Section 4.4, and conclude by pointing out limitations and potential improvements in Section 4.5.

[1]Implementation of walk learning is available at https://github.com/ppashakhanloo/CodeTrek.

4.1. Walk Policy

Representing programs by walks over relational graphs of code is challenging because the space of possible walks is combinatorially large. Random sampling will result in many unrelated walks among the sampled ones that not only do not contribute to improving the program representation, but also make it noisy. On the other hand, if walks are guided by a domain expert, the exploration space and noise decreases for the given task at the cost of significant engineering effort; nevertheless, some signals in large graphs may not be obvious to a human expert, preventing discovery of relevant and powerful relationships. Training a neural model to guide the walks can alleviate the undesirable consequences which are caused by human intervention.

An easy way to automate walk learning is to use a neural network policy that relies on walk history to recommend the next node to visit. This, however, poses a different challenge. The space of walks is discrete and combinatorial, thus no direct gradient update to the walk policy is available. Even after hypothetically addressing this challenge, it may still be difficult to interpret a learned walk-policy that is fully neural.

4.1.1. Learning Challenges

The challenges of walk learning arise from the fact that walks are latent—they are unobserved and lack supervision. However, a learned policy has the potential to improve the overall predictive performance of the model, and thus walk learning can be viewed as an expectation-maximization or E-M algorithm (Dempster et al., 1977). The E-step is to estimate the walk policy based on the current model, and the M-step is to improve the current model based on the estimated walk policy. The walk policy must now be designed in such a way that it can be estimated by using the Transformer (Vaswani et al., 2017).

4.1.2. Designing a Walk Policy

To design a walk policy, one can use some properties of relational graphs such as relation names, tuples, and key-foreign-key relationships to learn walk policies. The policy gives

each relation name a score (between 0 and 1) which is proportionate to the relevance of the relation to a specific task. These scores specify the next step in a random walk. For example, when the score of the `stmt` relation is 0.04 and the score of the `expr` relation is 0.02, it is twice as likely to step to a node of type `stmt` than a node of type `expr`, when there is a choice between the two as the next step of a walk. The policy is learned before training the actual model that solves the task. The policy learner updates the policy based on the aggregated relevance of relations in previous iterations of training.

Note that, although I present this work on learning walk policies in the context of relational graphs, there is nothing in the approach that makes it inapplicable to other graphs such as dataflow and control flow graphs constructed out of syntax trees, such as those used by Allamanis et al. (2018), or even basic-block graphs such as those used by Vasudevan et al. (2021). This approach works under the following conditions:

- A walk policy can be expressed in terms of bias over node types.
- The architecture of the program representation model that uses walks computes some notion of node importance such as attention scores. This notion can be used to learn an aggregate measure of importance about node types.

I present the results in the context of CodeTrek, but it would be applicable to other architectures such as Transformer-based systems (Hellendoorn et al., 2020) or graph-attention networks (Veličković et al., 2018).

4.1.3. Definition of Walk Policy

I define a *walk policy* over a relational graph as a mapping from every relation name to a score, indicating the relevance of each relation to the task for which the neural network is being trained. The score is a real number between 0 and 1. The scores are computed prior to training a model for the task. During the task training, they are used as guides for sampling walks over relational graphs.

4.1.4. Learning a Walk Policy

I revisit CodeTrek's model architecture (Pashakhanloo et al., 2022b) which is described in detail in Chapter 3. Consider a sequence of node type embeddings $\bar{\mathbf{e}}^{(n)} = [\mathbf{e}_1^{(n)}, ..., \mathbf{e}_N^{(n)}]$, edge type embeddings $\bar{\mathbf{e}}^{(e)} = [\mathbf{e}_1^{(e)}, ..., \mathbf{e}_{N-1}^{(e)}]$, and node value embeddings $\bar{\mathbf{e}}^{(v)} = [\mathbf{e}_1^{(v)}, ..., \mathbf{e}_N^{(v)}]$ that correspond to a walk w with a maximum of N nodes and $N-1$ edges, sampled from a relational graph G. The walk embedding $\mathbf{x}_w$ is computed as follows:

$$\mathbf{e}_w = \bar{\mathbf{e}}^{(n)} \parallel \bar{\mathbf{e}}^{(e)} \parallel \bar{\mathbf{e}}^{(v)} \in \mathbb{R}^{(3N-1)\times d} \tag{4.1}$$

where $\parallel$ is the concatenation operator. I elaborate on the details of the embedding in Section 4.1.5. I compute the self-attention weights for $\mathbf{e}_w$ as follows (Vaswani et al., 2017):

$$\text{Self-Attention}(\mathbf{e}_w) = \text{Attention}(\mathbf{e}_w W^Q, \mathbf{e}_w W^K, \mathbf{e}_w W^V) \tag{4.2}$$

where W^Q, W^K, and W^V are learned matrices, and

$$\text{Attention}(Q, K, V) = \text{Softmax}\big(QK^T/\sqrt{d}\big)V. \tag{4.3}$$

Intuitively, these weights are the results of comparing each element (node type, edge type, or value) in the walk to all the other elements in it. For more details about the attention mechanism refer to Section 2.1.

I compute the scores for relation names as follows. I discard all but the first N rows of Self-Attention($\mathbf{e}_w$) that correspond to the node types in walk w. I refer to this tensor as $A_w^{(n)} \in \mathbb{R}^{N\times d}$. I then expand each $A_w^{(n)}$ to a tensor with dimension R (the total number of relation names) so that the weights that correspond to each relation name have their own column, by aggregating the attention weights of all nodes with the same relation name. I

then compute a raw score vector S:

$$S = \text{Softmax}\big(\sum_{w \in W} A_w^{(n)}\big) \in \mathbb{R}^R \tag{4.4}$$

Each row in S corresponds to a relation name. At each training step, I update the scores:

$$S \leftarrow \gamma S + (1 - \gamma) S_{previous} \tag{4.5}$$

where $\gamma \in [0, 1]$ is a hyperparameter. Updates are repeated until the node types reach a fixed relative ranking. S is the list of scores that I use for guiding the random walks during the training process of a model for a given task. For example, if `S['stmt']` is 0.06 and `S['expr']` is 0.03, the learned walk policy enforces that the neighbors with node type `stmt` are 2x more likely to be visited by the random walk generator than neighbors with node type `expr`.

Stability of Ranking. I empirically observe that the relative rankings converge to a fixed point for all the tasks. Also, the rankings do not change across different training instances of the same task.

4.1.5. Embedding Graphs

To represent program graphs, I follow the neural representation I introduced in CodeTrek (Chapter 3) which outperforms GGNN, CuBERT, GREAT, and Code2Seq on bug-finding tasks. CodeTrek embeds each walk by treating it as a token sequence of relation names of its nodes, the edge types traversed, and their attribute values, in the order of traversal. Given each walk $w = [n_0, e_0, e_1, \ldots, n_{N-1}]$ consisting of N nodes and $N-1$ edges, an initial embedding X_w is produced:

$$X_w = X_w^{(n)} \parallel X_w^{(e)} \parallel X_w^{(v)} \in \mathbb{R}^{(3N-1) \times d} \tag{4.6}$$

where d is the embedding dimension and $\parallel$ represents concatenation, using three corresponding learnable embedding matrices E_n, E_e, E_v. The first segment, $X_w^{(n)}$, represents the

N node types (relation names), using an embedding lookup in $E_n \in \mathbb{R}^{R \times d}$, where R is the number of relations. The second segment, $X_w^{(e)}$, represents the $N-1$ edge types, using an embedding lookup in $E_e \in \mathbb{R}^{I \times d}$, where I is the number of KFK relationships in the database. Finally, the last segment, $X_w^{(v)}$, represents the attribute values of the N nodes; attribute values (using a V-sized WordPiece vocabulary for attribute values) are subtokenized, each subtoken is embedded using $E_v \in \mathbb{R}^{V \times d}$, and the subtoken embeddings are pooled into each node's attribute embedding. A sinusoidal positional encoding is computed for every segment of X_w. All encodings are concatenated to create the positional encoding of walk w:

$$\mathrm{P}_w = \mathrm{P}_w^{(n)} \parallel \mathrm{P}_w^{(e)} \parallel \mathrm{P}_w^{(v)} \in \mathbb{R}^{(3N-1) \times d} \tag{4.7}$$

Similar to CodeTrek, I use a Transformer encoder (Vaswani et al., 2017) to encode walk w as follows:

$$Z_w = \text{Transformer}(X_w + \mathrm{P}_w) \tag{4.8}$$

Over all the $3N-1$ elements of the walk on the last layer of Transformer, I perform attention pooling $f : \mathbb{R}^{(3N-1) \times d} \mapsto \mathbb{R}^d$ to obtain a d-dimensional walk embedding tensor $\mathbf{e}_w$.

4.2. Training

To use the introduced walk-policy learning technique, I train two models for each task. The first model takes a set of labeled graphs, each of which are represented as a set of uniform random walks and a label, (W, y). It learns a walk policy while optimizing for a specific task. The second model takes a set of labeled graph, each of which are represented as a set of walks that are sampled using the learned walk policy, and a label. In the current work, the hidden layers of the second model are similar to the first model, with an additional fully-connected network to train for predicting labels. Both models also embed each graph as a set of walk embeddings $\{\mathbf{e}_w\}_{w \in W}$ using a permutation-invariant operation. CodeTrek uses Deep Set architecture (Zaheer et al., 2017) for permutation invariance. I use the same cross-entropy loss function for both models and optimize them with Adam optimizer.

Task	Input	Description	Type
VarMisuse	Function and a Variable Access	Is the variable used correctly in the function?	Binary
VarMisuse-Fun	Function	Are all the variables used correctly?	Binary
VarShadow	Module	Is any of the global variables shadowed by any of local variables?	Binary
DefUse	Function and a Variable Definition	Is the variable definition used in the function?	Binary
DefUse-Fun	Function	Are all the variables defined in the function used after their definitions?	Binary
Exception	Module and a Masked Exception Type	What is the exception that the except statement should catch?	Multiclass
Exception-Fun	Function and a Masked Exception Type	What is the exception that the except statement should catch?	Multiclass
OpMisuse	Function	Are all the binary operators used correctly?	Binary

Table 4.1: Bug-finding Task Descriptions.

Task	# Training	# Validation	# Testing
VarMisuse	700,683	75,468	378,401
VarMisuse-Fun	700,683	75,468	378,401
VarShadow	70,183	21,794	39,845
DefUse	217,591	52,598	104,111
DefUse-Fun	33,182	8,149	16,296
Exception	18,456	2,086	10,334
Exception-Fun	18,456	2,086	10,334
OpMisuse	457,400	49,800	251,531

Table 4.2: The number of samples used for training, validation, and testing.

4.3. Evaluation

4.3.1. Experimental Setup

I evaluate the proposed approach to learn walk policies using five bug-finding tasks and their variants: VarMisuse, VarShadow, DefUse, Exception, and OpMisuse. I use the relational program graphs generated in Chapter 3 for training the first four tasks. The number of programs that are utilized to train, validate, and test these tasks is reported in Table 4.2. The description of the tasks and their variants is summarized in Table 4.1. These tasks are described in more detail in Chapter 3. I include a summary in this chapter for convenience. I use accuracy to measure the performance of all the tasks other than DefUse. I measure the performance of DefUse using ROC-AUC to account for its imbalanced dataset.

4.3.2. Effect on Accuracy

I compare the effectiveness of learned walk policy on performance by training three models with the same neural architecture but a different approach to scoring relations. In the first scenario, all relations are equally likely to be visited. In the second scenario, a human *expert* identifies a few relations as the most relevant ones for the task. Particularly, nodes with types `stmt`, `expr`, and `variable` are biased to be visited five times more often than others. In addition, in the EXCEPTION task, the score assigned to type `module` is 0 to avoid traveling from one function to another through their common `module` node. Thus, the walks will only lead to other functions using the call graph edges between them. In the final scenario, I employ the proposed policy learning mechanism to determine the relevance of relations without prior knowledge. The results of the models trained under these scenarios are reported in Figure 4.2. Models with learned policies consistently outperform models without policies by 6%–36% points. Interestingly, in all the tasks, models that use learned policies outperform the models for which a human expert scored the relations by 0.2–3.5% points.

The improved performance of models can be explained by examining the space of possible walks in each scenario. Choosing scores for walks reduces the space by prioritizing relations that are more relevant to solving a specific task, thus reducing the noise caused by irrelevant relations. To illustrate the usefulness of learning walk policies, consider the example in Figure 4.1. In this example, the model predicts the type of exception to catch on line 19. It must consider the exceptions raised by the `summarize` method on an instance of class `DiskQueue`. Hence, the definition of `summarize` (line 2) must be analyzed. In line 3, it calls `get_disk_cap` which raises an exception. This chain of function calls can be made explicit via a call graph, making it possible to walk along them.

Despite this, the walk space is still too large to be traversed at random. For instance, in Figure 3.7, there are millions of walks that start from the node that corresponds to the `except` statement. It is difficult to find walks that connect the `except` node to the `raise` node under a purely random scenario. One can reduce the exploration space by limiting the

exploration to walks that connect the two nodes through the body of the `try` block since the statements before and after that do not affect the caught exception. The challenge, however, is that only a domain expert can tune relation scores for a new task. Learning such scores eliminates the need for a domain expert for tuning. In Figure 3.7, the learned policy guides the walk generator to use the `except_successor` relation that connects the statements in a `try` block (e.g., line 16) to their corresponding `except` statement (e.g., line 19) more frequently. Having the walks visit `except_successor` during the traversal, reduces the number of 16-hop walks by over 95% compared to the unbiased case. Interestingly, this relation was overlooked by the human expert who scored the relations for the EXCEPTION task, justifying why learning the policy results in performing 2% points better than that of the human expert.

```
1  class DiskQueue:
2      def summarize(self):
3          dc = self.get_disk_cap()
4      ...
5
6      def get_disk_cap(self):
7          raise NotImplementedError
8
9  # many lines ...
10
11 class DiskQueueTest(QueueTest):
12     chunksize = 100000
13     self.q = DiskQueue()
14
15     def test(self):
16         try:
17             self.queue.summarize()
18             self.q.push('a')
19         except HoleException:
20             ...
```

Figure 4.1: Example EXCEPTION task.

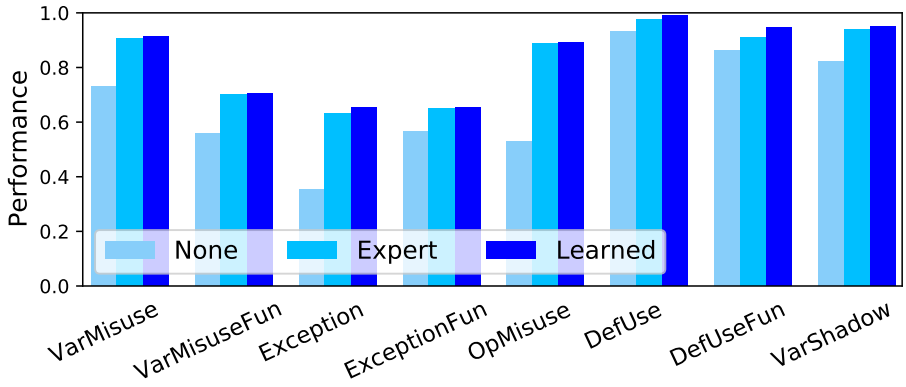

Figure 4.2: The effect of scoring relations on performance.

γ	0.0	0.2	0.4	0.5	0.6	0.8	1.0
Accuracy	0.52	0.60	0.61	0.65	0.57	0.56	0.36

Table 4.3: Sensitivity to γ variations.

4.3.3. Effect of Score Update Parameter

I examine the effect of changing γ during walk learning on the accuracy of the models that are trained to understand code. As described in section 4.1, this value represents the influence of the newly computed versus the previously computed scores during walk learning. The models learn the walk policy for the EXCEPTION task using varying values of γ and use them to train different models for the same task. As Table 4.3 shows, with γ set to 1, the model achieves the same accuracy as with uniform random walks. This outcome is expected because $\gamma = 1$ results in completely discarding the score values computed in the previous training iteration. At the other extreme, $\gamma = 0$ assigns an initially random but fixed score vector to relations throughout the policy learning. Regardless, it performs better than uniform random walks. Based on the empirical results, 0.5 is a reasonable value for γ in the selected tasks.

4.4. Interpreting Learned Policies

In this section, I take a closer look at the relations that have high scores. I qualitatively analyze some of the relations that are found relevant during the walk policy learning. The

Task	Most Relevant Relations
VarMisuse, VarMisuse-Fun	`stmt`, `scope`, `expr_context`, `expr`
VarShadow	`location`, `scope`, `variable`, `class`
DefUse, DefUse-Fun	`flow_bb_node`, `ssa_use`, `successor`, `scope`
Exception, Exception-Fun	`stmt`, `expr`, `variable`, `except_successor`
OpMisuse	`expr`, `cmpop_list`, `unnary_op`, `str`

Table 4.4: Relations with the highest learned scores.

top-4 relations with the highest scores are summarized in Table 4.4 for each task.

4.4.1. Example: Detecting Unused Definitions

For DefUse, I compare the relations with the highest learned scores with a CodeQL query crafted for the same purpose. `flow_bb_node`, `successor`, and `ssa_use` are the highest-ranking relations for this task, and all three are crucial for the CodeQL query. The first two relations establish control flow relationships between basic-blocks and statements, and the third relation computes dataflow information.

4.4.2. Example: Detecting Shadowed Global Variables

VarShadow is another example—a global task which requires inspecting inner scopes (possibly multiple classes) for variables that shadow global variables. Interestingly, the learned policy detects that `class` is one of the relations with the highest scores.

4.5. Discussion

While the proposed policy learning mechanism is effective, it has limitations and can be improved. Some of the shortcomings are highlighted in this section.

Scope of the Policy. One could argue that a walk policy that only scores relations might not be sufficient for some tasks. Consider the following example. In a code database schema, there are key-foreign-key relationships between `func` and both of `flow_bb_node` and `call` relations. This means that there is an edge between a node of type `func` and a node of

type `flow_bb_node`. There is also an edge between a node of type `func` and a node of type `call`. Traversing the `call-func` edges can be interpreted as visiting other functions that a function calls. The `flow_bb_node-func` edges, on the other hand, connect `function` nodes to the root nodes of control-flow graphs of functions' bodies. Depending on whether the task requires inter-procedural (global) or intra-procedural (local) analysis, the relevance of the `func` relation varies. For such scenarios, I conjecture that scores for *edge types* should be learned rather than relations types. An interesting future exploration is incorporating other factors such as edge types, values, and the length of walks into the learned walk policy.

Memory. The current mechanism for learning walk policies also lacks memory, which might hinder its ability to capture complex patterns.

Anchors. Another piece of information that should be specified by an expert is the anchor node. In a practical approach, every walk could start from the root (i.e., the top-level node in the program graph such as a `module` node) and learn better anchor nodes from there. There are also open questions regarding anchors; how many nodes should be selected as anchors? Should all the anchor nodes for a particular task have the same relation name?

4.6. Summary

In this chapter, I presented a deep learning approach for learning walk policies over relational graphs. I showed that sampling walks using a learned walk policy can result in models with better performance than that of a policy designed by a human expert. Among the opportunities for future research are: (1) incorporating edge types, values, and other characteristics of walks into the learning of walk policies, (2) expanding the application of the proposed policy learning mechanism beyond code understanding, and (3) providing walk policies with memory (state) to capture more complex patterns.

CHAPTER 5

Robust and Interpretable Code-to-Text Translation

High-quality source code documentation is essential for understanding a program and maintaining it (Roy et al., 2021). Despite the importance of documentation, developers may fail to document their code because writing clearly and concisely is difficult. A lack of a step-by-step process to follow for drafting code descriptions may contribute to this difficulty. Documenting code is also challenging because best practices are unclear, making it one of the most challenging tasks developers encounter.

The difficulty of writing source code descriptions has motivated researchers to train deep neural networks to automate this task. These networks consume source code and produce a description in text format. As a result, they are often called code-to-text neural models.

Code-to-text neural models have made remarkable advances since their inception. While machine learning algorithms exhibit promise and power in code comprehension tasks, they have limitations. In particular, there is a lack of robustness and a lack of interpretability (Du et al., 2019). Often, the user is not aware of the internal logic or inner workings of these models. It prevents humans from interpreting or understanding how particular decisions are made by the system (Montavon et al., 2017).

Robustness is a model's ability to not change its predictions when it undergoes semantic-preserving code transformations such as renaming variables, adding or removing deadcode, or exchanging `for` loops with `while` loops. Recent findings show that deep neural networks are brittle to data changes (Gao et al., 2020; Ko et al., 2019; Garg and Ramakrishnan, 2020; Carlini and Wagner, 2017). For instance, they are vulnerable to small input perturbations (Henke et al., 2022).

An *interpretable* model is one which can be used to explain to humans—of course, in understandable terms—why a particular prediction was made given specific inputs (Hall et al.,

2017). An interpretable model is transparent and thus easier to trust (Thampi, 2022). For AI to succeed as a reliable tool for decision support, it must not only provide the final predictions; it must also communicate in a way that allows a human to use intuition and reasoning to their fullest extent (Gilpin et al., 2018; O'neil, 2016). In contrast to accuracy, interpretability is a qualitative factor, making it harder to evaluate. Perhaps this explains why it is often overlooked.

To build an effective code-to-text model to perform a task such as code summarization task—where a short natural language description is generated for the input source code—the code representation must capture program semantics comprehensively (Ma et al., 2022). The existing models rely on Abstract Syntax Tree (Alon et al., 2018) or source token sequences (Guo et al., 2020b; Hu et al., 2018b; Wei et al., 2020; Xie et al., 2021b; Lu et al., 2021; Feng et al., 2020) to embed programs for code summarization tasks. However, these approaches fail to capture the semantics of programs completely.

In this chapter, I elaborate how CodeTrek can be used to train robust and interpretable models for automating source code summarization. According to my findings in Chapter 3 and Chapter 4, CodeTrek can address the robustness and interpretability challenges in code-to-text tasks by relying on learned sampled walks over semantically-rich relational program graphs. I show the effectiveness of CodeTrek over two well-known baseline approaches, namely, CodeBERT and Code2Seq.

Chapter Organization. I define the key terms, namely robustness and interpretability, in Section 5.1, and elaborate on the implementation and training process in Section 5.2. Then, I evaluate the effectiveness of CodeTrek in terms of performance and robustness for code summarization in Section 5.3. Additionally, I discuss the interpretability of CodeTrek through case studies in Section 5.4. In the end, I discuss limitations and improvements that can be made in the future (Section 5.5).

5.1. Key Definitions

5.1.1. Robustness

Robustness is a model's ability to not change its predictions when it undergoes small input perturbations. The definition of *input perturbations* in the context of source code differs from that found in natural language processing or computer vision; it is imperative to guarantee that changes made to source code follow syntax rules. Also, one must ensure that the code-text pairs remain valid after perturbing the code. The field of program analysis introduces source code perturbations that follow syntax rules and are guaranteed to preserve semantics. As a result, input perturbations in the context of source code are semantic-preserving code transformations such as renaming variables, adding or removing dead code, or exchanging `for` loops with `while` loops. Recent findings show that deep neural networks are vulnerable to these semantic-preserving code transformations (Henke et al., 2022).

5.1.2. Interpretability

An *interpretable* model is one which can be used to explain to humans why a particular prediction was made given specific inputs (Hall et al., 2017). It is difficult to formalize interpretability because it is subjective. Moreover, interpretability is domain-specific, so it cannot be defined universally. Depending on the context, different types of explanation might be useful (Carvalho et al., 2019). For more details on these definitions please refer to Section 7.9. This need for interpretability arises because for some problems or prediction tasks, it is not sufficient to know *what* has been predicted (Doshi-Velez and Kim, 2017). Because a correct prediction only partially solves the original problem, the model must also explain *how* it reached the prediction. For the purpose of this work, I employ the definition suggested by Zhang et al. (2021): "Interpretability is the ability to provide explanations in understandable terms to a human". In this definition, *understandable term* refers to the domain knowledge related to the task. For example, Linux operating system source code is interpretable under this definition, although it might be overwhelming for a developer (Zhang et al., 2021).

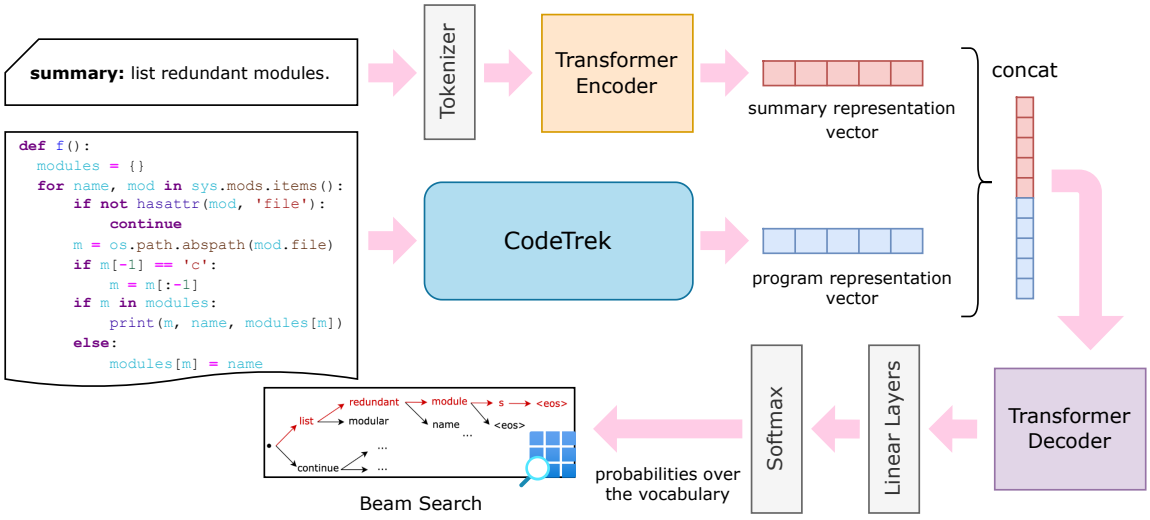

Figure 5.1: Code summarization with CodeTrek.

Intrinsic Interpretability. A model that is intrinsically interpretable is one that can be interpreted on its own. Constraints derived from domain knowledge can be applied to a model to achieve this property. For example, CodeTrek models can only traverse nodes in the relational graph that are linked through referential integrity constraints. Intrinsic interpretability is also called transparency and answers the question of how the model works (Lipton, 2018).

5.2. Implementation

I implement an instance of abstractive source code summarization using CodeTrek. In this method of summarization, words are chosen according to the semantics of the code snippet. The summary may include words not found in the source.

Since CodeTrek is just an encoder, I put a standard Transformer decoder on top of it to enable code summarization. The decoder stack consists of 6 layers. The loss function I use for the decoder is cross-entropy loss. Then, the language modeling head follows which consists of two linear layers that link the decoder to the softmax layer which is responsible for the production of the next token probabilities over the vocabulary of size 50,348 tokens. Figure 5.1 illustrates code summarization with CodeTrek. The abstract blue component

which is labeled CodeTrek is responsible for embedding the source code. The details of this component can be found in Section 3.2. During the training, I compute the cross-entropy loss after the softmax layers. Also, I use walk policy learning (Chapter 4) to effectively guide the walk generation. During inference, I do not consider the summary as the input since it is the goal but I do a beam search with a width of 1 to generate the summary. Also, I set the learning rate to 5e-5, the batch size to 16, and the decoder target length to 128[1].

5.3. Evaluation

I elaborate on the experimental setup and evaluation results in this section. The main goals of these experiments are answering the following questions.

1. What is the **performance** of CodeTrek in the code summarization task?
2. In the context of code summarization, how **robust** are CodeTrek's predictions?
3. How **sensitive** is CodeTrek's performance to the length of the sampled walks?

5.3.1. Experimental Setup

Code Summarization. Given a function whose name is masked, code summarization automatically generates a short natural language description that characterizes the input source code. This task is described as follows in the task specification language which is introduced in Section 3.3.

```
{
    RB = { "stmt"; "var"; "expr"; "func"; ... },
    RQ = { },
    S = {
        C = { x : (x instanceof "func") and
                  (x.name is "MASK")},
        B = {  },
        min = 10,
        max = 16
    },
    N = 50
}
```

[1]I adopted CodeBERT's proposed architecture for code-to-text translation, borrowed from https://github.com/microsoft/CodeXGLUE/tree/main/Code-Text/code-to-text.

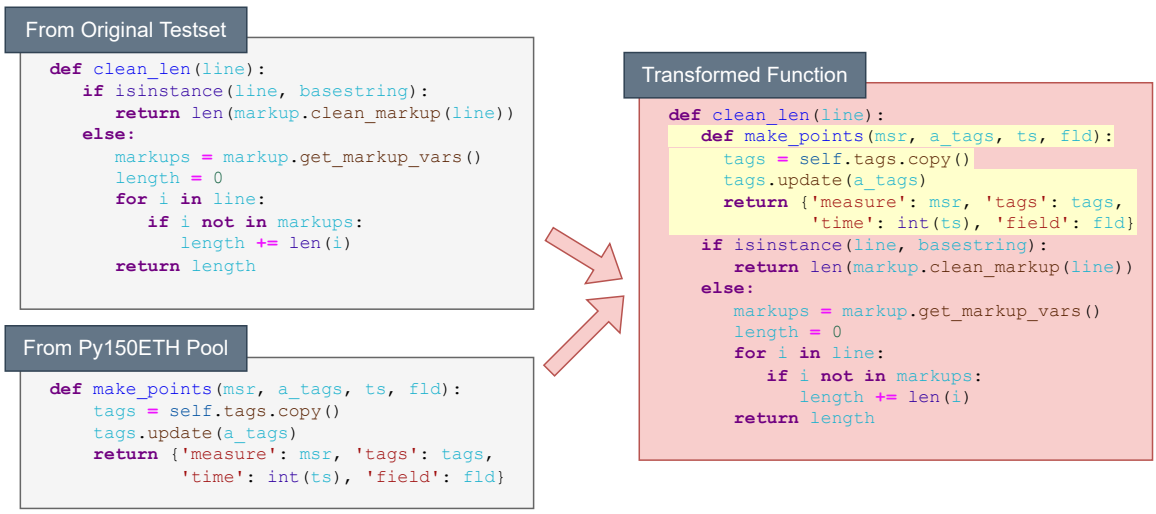

Figure 5.2: The process of adding deadcode to transform the test set.

#Training	#Validation	#Testing
250,720	13,713	14,418

Table 5.1: Code Summarization dataset size.

Dataset. In order to empirically evaluate the performance of the code summarization task, I used the datasets from the CodeXGLUE benchmark (Lu et al., 2021). Specifically, I use CodeSearchNet (Husain et al., 2019), which contains thousands of Python code snippets and their short natural language descriptions. Each sample in this dataset is represented using a code-comment pair. Table 5.1 reports the dataset statistics. To conduct the robustness experiments in section 5.3, I generate a perturbed, yet semantically-equivalent test set from the original test set. To do so, for each function in the test set, I randomly select a Python function from Py150ETH dataset and insert it before a random statement in the function's body. An example of this process is shown in Figure 5.2. I further replace all the function names by `MASK` to ensure the models generate the summaries only based on the arguments and the function body.

Baseline Models. I use two well-known models for code summarization as baseline models, namely, Code2Seq and CodeBERT. Code2Seq (Alon et al., 2018) is an encoder-decoder framework. Its encoder receives a vector of leaf-to-leaf paths, where paths are sampled from

the Abstract Syntax Tree, and the decoder outputs the tokens that build the summaries. CodeBERT (Feng et al., 2020) is a 125-million parameter, pre-trained, encoder-only model which follows an architecture similar to that of BERT (Devlin et al., 2018). CodeBERT encodes both source code and natural language text. Since CodeBERT does not have a decoder—similar to CodeTrek—I augment the architecture with a standard transformer decoder for generating the code summaries.

Evaluation Metrics. To evaluate the effectiveness of CodeTrek and baseline models in code summarization task, I use three metrics: BLEU, ROUGE-L, and METEOR. These metrics measure token overlap between the prediction and the reference. Using multiple evaluation metrics can be a good idea when evaluating text summarization systems, as different metrics may have different strengths and weaknesses. BLEU, ROUGE-L, and METEOR are all commonly used metrics for text summarization evaluation, and each has its own advantages and disadvantages.

BLEU (Papineni et al., 2002) is the most common evaluation metric used for assessing the effectiveness of code summarization task. It works by comparing n-grams in the predicted and reference summaries. In particular, following the evaluation methodology proposed by Iyer et al. (2016), I use BLEU-4 which measures the average n-gram precision on a set of reference sentences with a penalty for overly short sentences. I also apply +1 smoothing (Lin and Och, 2004) before reporting the results. In this smoothing strategy, 1 is added to both the numerator and denominator. The computations are shown in Equation 5.1 where $P_{n,t,r}$ is the geometric mean of n-gram precisions (Equation 5.2) and $BP_{t,r}$ is the brevity penalty (Equation 5.4). Each p_n in Equation 5.2 is computed as Equation 5.3 where m_n is the number of matched n-grams between translation t and reference r, and l_n is the total number of n-grams in translation t.

$$BLEU_{n,t,r} = P_{n,t,r} \times BP_{t,r} \tag{5.1}$$

$$P_{n,t,r} = \big(\prod_{i=1}^{n} p_i\big)^{\frac{1}{n}} \tag{5.2}$$

$$p_i = \frac{m_i}{l_i} \tag{5.3}$$

$$BP_{t,r} = min\Big(1, exp(1 - \frac{|r|}{|t|})\Big) \tag{5.4}$$

The brevity penalty punishes the score if the translation is much shorter than the reference.

ROUGE-L (Lin, 2004) is similar to BLEU but it also incorporates the *recall* between candidate and reference strings. Computing ROUGE-L relies on the notion of Longest Common Subsequence (LCS). LCS takes into account sentence-level structure similarity. ROUGE-L is computed as Equation 5.5 where R_{lcs} and P_{lcs} are LCS-based recall and precision as shown in Equation 5.6 and Equation 5.7.

$$ROUGE_L = \frac{(1+\beta^2)P_{lcs}R_{lcs}}{\beta^2 P_{lcs} + R_{lcs}}, \beta = 1.2 \tag{5.5}$$

$$R_{lcs} = \frac{|LCS(O_p, O_r)|}{|O_r|} \tag{5.6}$$

$$P_{lcs} = \frac{|LCS(O_p, O_r)|}{|O_p|} \tag{5.7}$$

Here, O_p is the predicted summary and O_r is the reference summary.

Finally, METEOR (Banerjee and Lavie, 2005) evaluates how well the generated comments capture content from the references via recall and precision. METEOR's matching criteria support not only word matches that are identical in each string, but also matches that are simple morphological variants of each other (e.g. they have the same stem) or synonyms. METEOR is calculated as:

$$METEOR = (1 - \gamma f^{\beta}) \frac{PR}{\alpha P + (1-\alpha)R} \tag{5.8}$$

where P and R are unigram precision and recall, and f is a fragmentation fraction. The fragmentation fraction accounts for the order of unigrams that appear in translations. I follow the default values for the parameters which are suggested by Banerjee and Lavie

	BLEU (%)	ROUGE-L (%)	METEOR (%)
CodeTrek	20.17	57.78	15.09
CodeBERT	19.84	51.13	14.10
Code2Seq	18.73	49.73	9.95

Table 5.2: Performance of models on code summarization task for the *original* dataset.

(2005): $\alpha = 0.9$, $\beta = 3$, and $\gamma = 0.5$.

Metrics and Human Judgement. BLEU is simple to compute and has been widely used in the literature. However, it has been criticized for being overly dependent on exact word matches and not taking into account semantic equivalence (Papineni et al., 2002; Banerjee and Lavie, 2005; Lin, 2004). I choose to include this metric due to its wide use in the literature. ROUGE-L, is simple to compute and has been shown to have a better correlation with human judgement than BLEU (Lin, 2004). ROUGE-L is recall-oriented which means it is more focused on the content that is included in the generated summary, rather than what is missing. However, ROUGE-L does not take into account synonymy or stemming. This semantic limitation is addressed in METEOR at the cost of becoming a computationally expensive metric as it requires aligning the generated summary with the reference summary.

5.3.2. Performance

I use three machine translation metrics, BLEU, ROUGE-L, and METEOR, to compare the performance of the CodeTrek model in generating code summaries with two well-known baseline models, CodeBERT and Code2Seq. Table 5.2 reports the performance results of all these models. CodeTrek outperforms both of the baselines. More specifically, CodeTrek improves upon the baselines by 0.33–1.44% points on BLEU, 6.65–8.05% points on ROUGE-L, and 0.99–5.14% points on METEOR. **The conclusions reached by all three metrics are the same:** The experimental results suggest the importance of using semantically-rich source code graphs to represent programs.

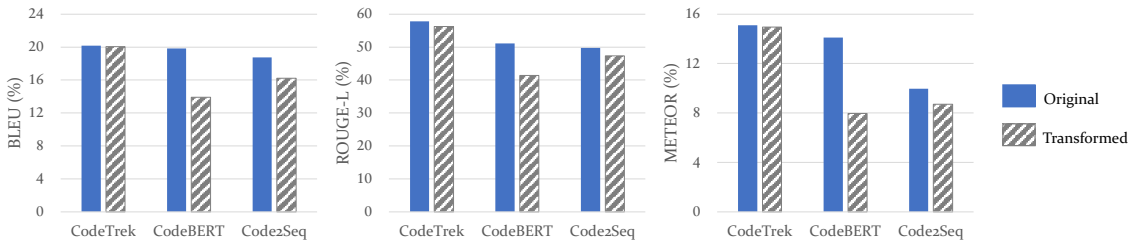

Figure 5.3: The performance of CodeTrek before and after transforming the test set.

5.3.3. Robustness

I measure the robustness of CodeTrek against semantic-preserving code transformations and compare it with the baseline approaches. To create a transformed test set, I randomly add a function definition inside the function body. This serves as adding deadcode, and is therefore a semantic-preserving change. Figure 5.2 illustrates an example of such a transformation. Then, I evaluate the baseline models and CodeTrek model for code summarization using the transformed test set. The comparison between the performance of CodeTrek before and after transforming the test set with respect to BLEU, ROUGE-L, and METEOR is reported in Figure 5.3.

The experimental results show several significant findings. In the first place, when using the transformed test set instead of the original test set, CodeTrek's performance degrades significantly. Specifically, the performance drops by 0.12% points on BLEU, 1.55% points on ROUGE-L, and 0.13% points on METEOR. This suggests that CodeTrek is robust against semantic-preserving code transformations. Second, consistent with the results reported in subsection 5.3.2, CodeTrek outperforms the baseline models on all three evaluation metrics. Third, the runner-up in this experiment is Code2Seq—another walk-based approach. This result suggests that sampling walks can be a promising strategy for robustness. Finally, CodeBERT's performance drops significantly on the transformed test set. Specifically, the performance drops by 5.92% points on BLEU, 9.73% points on ROUGE-L, and 6.15% points on METEOR. In line with recent studies (Mukherjee et al., 2021), relying only on token sequences may not be sufficient for ensuring robustness in code comprehension models.

The poor performance of the baselines can be explained by the fact that the transformed test set is out-of-distribution. While this is the case, CodeTrek's performance is explained by the inductive bias generated by rich relational information during training.

An astute reader may wonder why Code2Seq performs poorly in robustness despite its similarity to CodeTrek in being a walk-based technique. An explanation can be found by comparing the original and transformed abstract syntax trees. Even a single program transformation may affect several leaf-to-leaf paths and lead to a drastically different prediction. This effect has also been observed by others (Henke et al., 2022). Changes of this nature have less of an impact on relational program graphs that CodeTrek uses.

Example: Summaries before and after the Transformation. The following example helps illustrate the the behavior of CodeTrek and baseline models after the code is transformed. The code snippet in Figure 5.4 is the original form of a function that sends a guess to server. All the models generate reasonable summaries for this function. CodeTrek's prediction and CodeBERT's prediction is *write guess to server* and Code2Seq's prediction is *write to server*.

```
def MASK(guess):
  if connected():
       gui.status_bar_info("Query...", False)
       try:
            os.write(guess.to_bytes())
            os.flush()
       except Exception:
            gui.status_bar_info("Failed.", True)
```

Figure 5.4: Original function for *sending a guess to server*.

Upon transforming the function in Figure 5.4 by adding a random dead function to obtain the function in Figure 5.5, CodeTrek still predicts the same description but the two baselines fail to generate meaningful results. In this function, the lines 2–8 are deadcode but CodeBERT and Code2Seq do not *understand* this notion. The generated summary by CodeBERT is *start server gui* and the summary by Code2Seq is *guess host handler*.

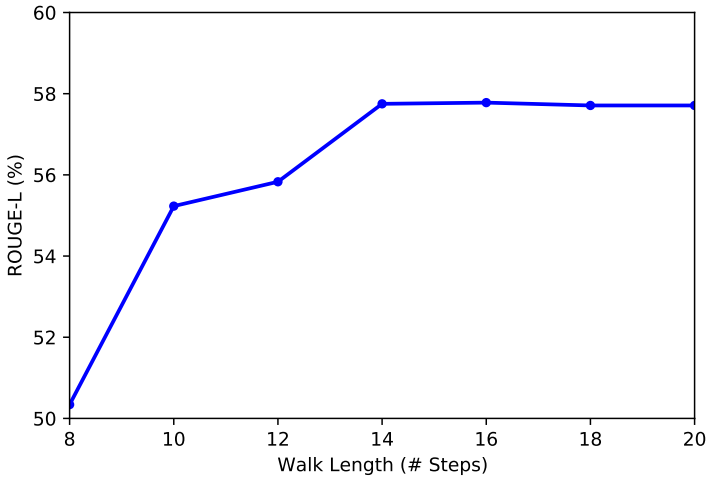

Figure 5.6: Sensitivity to the length of walks in code summarization task.

```
def MASK(guess):
  def start_server(handler, host, port):
       msg = Message()
       httpd = ThreadedHTTPServer((host, port), handler)
       try:
            httpd.serve_forever()
       finally:
            httpd.server_close()

  if connected():
       gui.status_bar_info("Query...", False)
       try:
            os.write(guess.to_bytes())
            os.flush()
       except Exception:
            gui.status_bar_info("Failed.", True)
```

Figure 5.5: Transformed version of the function in Figure 5.4.

5.3.4. Sensitivity to Walk Lengths

To measure the sensitivity of CodeTrek to the length of walks, I train a number of models for the code summarization task with walks of length 8–20 steps. I report performance changes in Figure 5.6.

A longer walk tends to improve performance. Walks that are shorter than 10 steps result in a model with low performance. For example, ROUGE-L is 50.34% when CodeTrek is trained with walks of length 8. The reason is that such short walks are unable to capture enough semantics for making informed predictions. As the number of steps increases, the performance improves, and when the number of steps is 16, the model achieves its highest performance which is 57.78% on ROUGE-L. It appears that this general trend of performance improving with more steps slows down at about 14 steps when extending the walks by 2 steps only contributes 0.03% points to the improvement of performance. An increase in the length of the walks does not improve performance any further. In fact, increasing the walk length from 16 to 18 results in a small drop of 0.07% points on ROUGE-L.

5.4. Qualitative Study of Interpretability

In this section, I qualitatively assess the interpretability of CodeTrek's predictions using several case studies. As explained in Section 3.2.6, CodeTrek allows inspecting the individual walks that contributed the most to predictions to see how it aligns with human reasoning. I refer to these walks as the *explanation* that CodeTrek produces along with the original prediction. In particular, I observe that the walk scores follow this pattern: k top walks have significantly higher scores than the rest of the walks. For instance, in Case Study 1, four walks have scores of nearly 0.9 out of 1.0, and the rest of the walks have scores of less than 0.1. I refer to the former as explanations. In each case, I examine the explanations and show how they pertain to each piece of the generated function summary. In Case Study 1 (Section 5.4.1), I explain how walks correspond to the generated summary. Case Study 2 (Section 5.4.2) exemplifies the case where interpretability helps debug an incorrect prediction. Finally, Case Study 3 (Section 5.4.3) shows how CodeTrek's partial summaries are nonetheless reliable.

5.4.1. Case Study 1

I start by explaining how the guided random walks that CodeTrek uses to embed source code correspond to the final natural language summary that is generated. The code snippet

listed in Figure 5.7 takes the name of an environment variable as input and returns a filename that is stored in it. The reference summary for this Python function is *get the name of the file from environment variable*. CodeTrek correctly generates *get filename from the given variable* as the function's summary. Next, I inspect two guided random walks that CodeTrek generates for embedding the code snippet. These walks obtain the highest walk scores so I refer to them as explanations for this prediction. These walks are illustrated in Figure 5.8.

```
def MASK(variable):
    filename = os.environ.get(variable)
    if filename is None:
        msg = f"Environment Var:  variable"
        raise EnvironmentError(msg)
    return filename

# Reference:
# get the name of the file from environment variable.
# CodeTrek:
# get filename from the given variable.
```

Figure 5.7: Code snippet and CodeTrek's prediction for Case Study 1.

The anchor (i.e., starting node) in both of the walks in Figure 5.8 is a `func` node that corresponds to the function definition node in the relational program graph. Walk number 1 connects the variable `filename` to the call that is made to the `get` function. Particularly, there is a `stmt` node of kind `assign` which corresponds to assigning the result of the call to the `get` function to `filename`. Upon manual inspection, this walk clearly corresponds to *getting filename*. Walk number 2 is a simpler walk. It connects the variable `variable` to a node of type `param`. Therefore, this walk makes up the *given variable* portion of the summary. The combination of these two walks explain why the generated summary is *get filename from the given variable*.

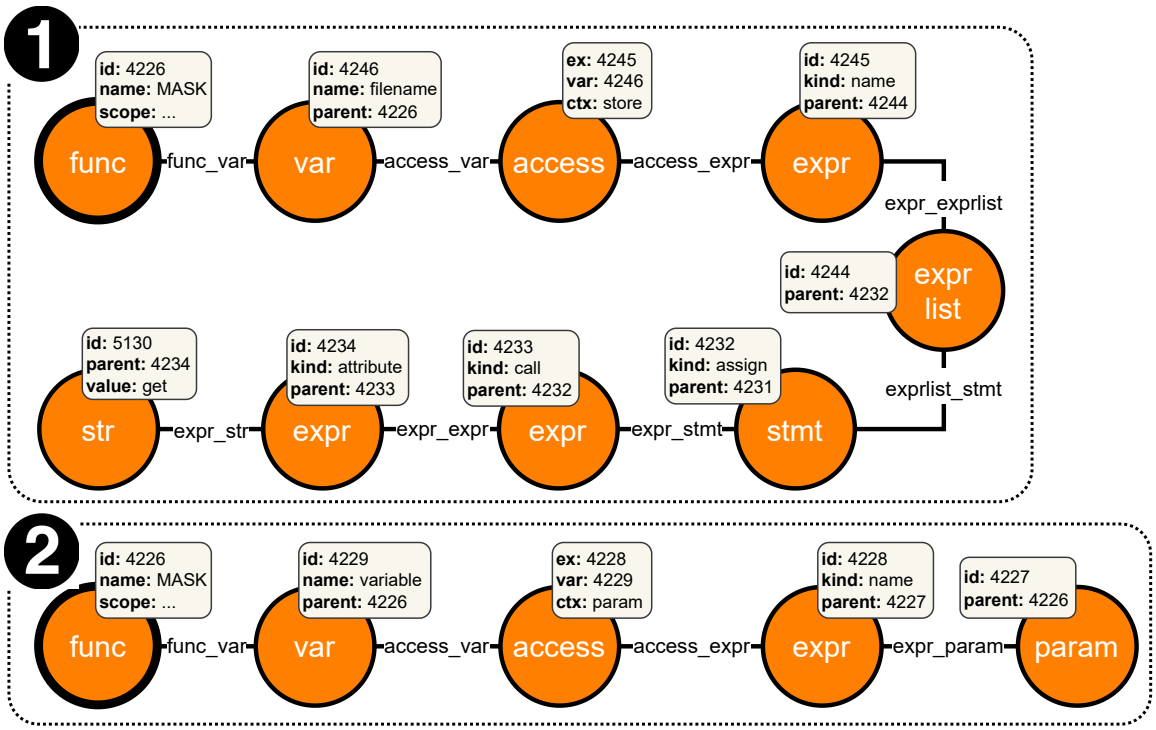

Figure 5.8: Most important walks for the code snippet in Figure 5.7.

5.4.2. Case Study 2

In this case study, I discuss one of the most significant benefits of interpretability in neural models of code: debugging. In fact, interpretability is a useful debugging tool for finding problems and biases in data (Molnar, 2020). In particular, I discuss an example where CodeTrek produces an incorrect summary. Nevertheless, the walks with the highest score help explain where the prediction is wrong.

The code snippet listed in Figure 5.9 takes an object as input, iterates over all the members of it, finds the corresponding values, and returns all the members of the object as `(member, value)` pairs. The reference summary for this Python function is *return all members of an object as pairs*. CodeTrek generates *return directory of a given object* as the function's summary which turns out to be incorrect. Now, I inspect the four guided walks that CodeTrek generates for embedding the code snippet. These walks, illustrated in Figure 5.10, obtain the highest walk scores so I refer to them as explanations for this prediction.

The anchor (i.e., starting node) of all walks in Figure 5.10 is the `func` node, which corresponds to the function definition node in the relational program graph. The first walk connects the variable `object` to a `param` node. Thus, this walk constitutes the *given object* portion of the summary. Walk number 2 connects the variable `results` to a node of type `stmt` that corresponds to the `return` statement on line 11. A domain expert can clearly argue that this walk corresponds to a natural language description such as *return results*. Walk number 3 corresponds to line 3 of the code snippet (which is emphasized by visiting the `scope` node after the `func` node), and corresponds to the call made to a built-in function `dir`. An `expr` node of kind `call` shows this call. Walk number 4 is a longer walk that connects the mentioned call to its input argument, i.e., a `var` node that corresponds to variable `object`. These two walks correspond to a natural language description of the form *get object directory*. The combination of all these walks explains why the generated summary is *return directory of a given object*.

The troublesome part of the prediction made by CodeTrek is the presence of *directory* instead of *all members* in the summary. Intuitively, this part of the summary corresponds to walk number 3 that tentatively gives the natural language description of *get directory* to the call to `dir` function. Upon inspecting the training dataset, I found that the keyword `dir` has been used in 1052 Python functions. In 658 cases, `dir` refers to a directory. In the rest of the cases, a call has been made to the built-in `dir` function which "returns all properties and methods of the specified object, without the values"[2]. Manual inspection of these 394 cases shows that only in 7 cases the docstring points to the notion of "members". This shows a bias in the data. One can use this piece of debugging information to improve the distribution of the training data.

[2]https://docs.python.org/3/library/functions.html#dir

```python
def MASK(object):
    results = []
    names = dir(object)

    for key in names:
        try:
            value = getattr(object, key)
        except AttributeError:
            value = object.__dict__[key]
        results.append((key, value))
    return results

# Reference:
# return all members of an object.
# CodeTrek:
# return directory of a given object.
```

Figure 5.9: Code snippet and predictions for Case Study 2.

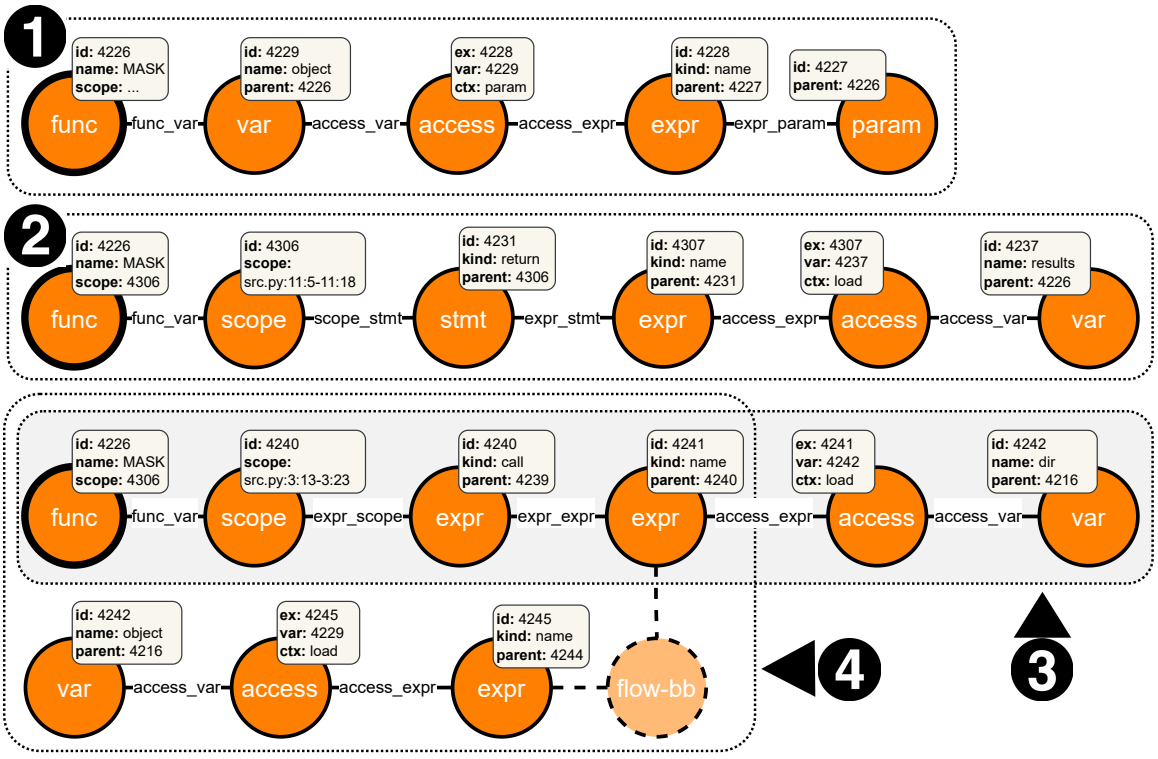

Figure 5.10: Most important walks for the code snippet in Figure 5.9.

5.4.3. Case Study 3

In this case study, I discuss a scenario where all the models (baselines and CodeTrek) fail to make correct predictions. Nevertheless, in contrast to the baselines, CodeTrek manages to generate a partially correct summary without incorrect information in it.

The code snippet listed in Figure 5.11 takes a number, a format, and a locale as input. It then applies the appropriate format based on the given locale to the given decimal number. The reference summary for this Python function is *format the given decimal number*. CodeTrek generates *apply given locale* which is only partially correct, CodeBERT generates *set locale to pattern* which is incorrect, and Code2Seq generates *is number valid* which is also incorrect. Now, I inspect two guided random walks that CodeTrek generates for embedding the code snippet. These walks obtain the highest walk scores so I refer to them as explanations for this prediction. These walks are illustrated in Figure 5.12.

```
def MASK(number, format, locale):
    locale = locale.parse(locale)
    if not format:
        format = locale.decimal_formats.get()
    pattern = parse_pattern(format)
    return pattern.apply(number, locale)

# Reference:
# format the given decimal number.
# CodeTrek:
# apply given locale.
# CodeBERT:
# set locale to pattern.
# Code2Seq:
# is number valid.
```

Figure 5.11: Code snippet for Case Study 3.

The anchor (i.e., starting node) in both walks in Figure 5.12 is a `func` node that corresponds to the function definition node in the relational program graph. Walk number 1 connects the variable `locale` to the call that is made to the `apply` function. The call is specified

with a node of type `expr` whose kind is `call`. Upon manual inspection, this walk appears to correspond to a description such as *apply locale*. Walk number 2 is a simpler walk. It connects the variable `locale` to a node of type `param`. Therefore, this walk makes up the *given locale* portion of the summary. The combination of these two walks explain why the generated summary is *apply given locale* which is partially correct. In contrast to CodeTrek, CodeBERT and Code2Seq generate summaries that are incorrect and can mislead a developer. In particular, CodeBERT generates a summary that does not consider the flow of the statements in the program at all, and Code2Seq generates a completely arbitrary one.

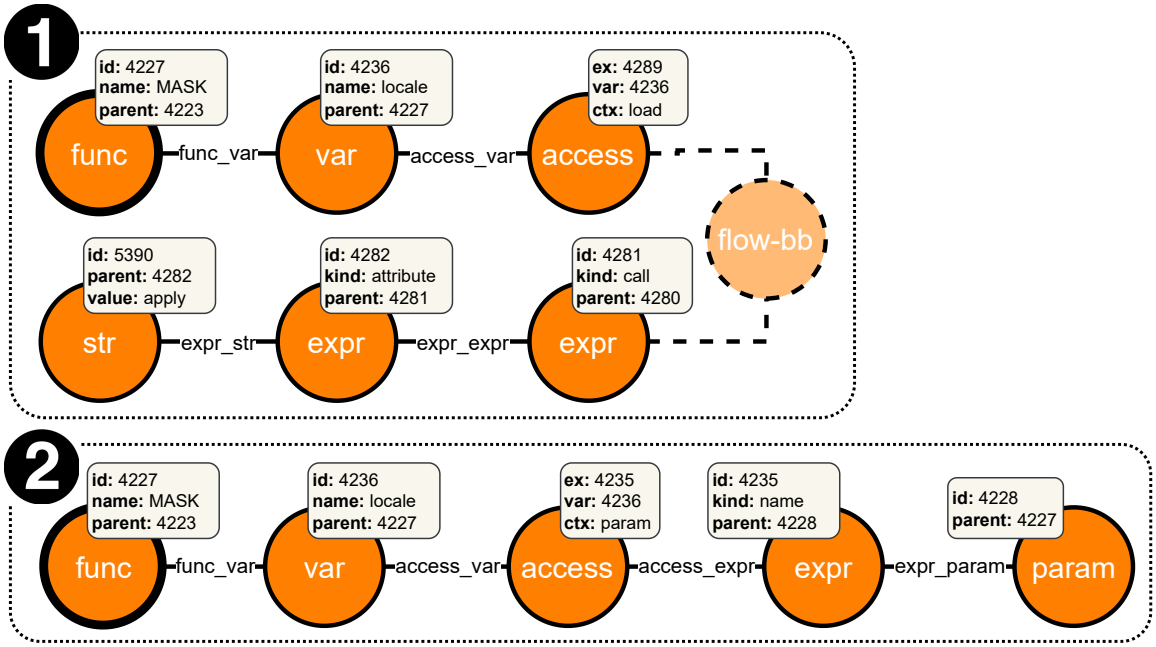

Figure 5.12: Most important walks for the code snippet in Figure 5.11.

5.5. Discussion

Meaningfulness of Relations. In previous chapters, I argued that CodeTrek does not fundamentally rely on Semmle's CodeQL, but is applicable to any relational database of code. The interpretability of CodeTrek, however, is dependent upon the meaningfulness of the database relations. Consequently, if relational databases contain arbitrarily defined relationships without semantic meanings, interpretability is not achieved.

Quantifying Interpretability. In this chapter, I discussed interpretability only qualitatively and through case studies. Quantifying interpretability in deep learning can be challenging since the concept is complex and multifaceted. The best way to do it depends on the specific task, the type of model, and the resources available. Nevertheless, there are various interpretability metrics that have been proposed in the literature for domains other than code-understanding (Hooker et al., 2019), such as feature importance measures (Lin and Gao, 2022; Lundberg and Lee, 2017; Ribeiro et al., 2016), saliency maps (Simonyan et al., 2013), and decision trees (Quinlan, 1987). These metrics provide a quantitative measure of how much a model's decision is influenced by certain inputs or features. Note that none of these approaches are free of criticisms.

A *feature importance measure* quantifies the contribution that each feature or input variable makes to the prediction of a machine learning model. They identify which input variables are most significant in determining the model's predictions (Ribeiro et al., 2016). *Saliency maps* are used to explain the predictions of deep neural networks by highlighting the parts of an input that are most significant for a given prediction. The idea behind saliency maps is to create a heatmap of the input, where each pixel is colored based on its importance for the model's prediction (Simonyan et al., 2013). Finally, *decision trees* can be used as an explanation of the model's predictions, by breaking down the decision process step by step, and showing how different features or variables contribute to the final prediction.

5.6. Summary

In this chapter, I showcased the effectiveness of CodeTrek in code summarization task by conducting a number of empirical studies on a public, commonly-used corpus of code-comment pairs in addition to qualitative studies. The results showed that CodeTrek outperforms CodeBERT and Code2Seq by 0.33—1.44% points on BLEU, 6.65—8.05% points on ROUGE-L, and 0.99—5.14% points on METEOR. Moreover, I showed that CodeTrek is robust against out-of-distribution data which was presented to the model in the form of a transformed test set. Finally, I qualitatively showed the promise of CodeTrek framework in producing predictions that are interpretable.

CHAPTER 6

Future Work

In this dissertation, I studied the integration of declarative program analysis with deep neural networks. My research showed that program analysis combined with neural networks can improve the performance of various code understanding tasks. These results laid the foundation for future research. In future work, I plan to investigate further ways to leverage the strengths of both declarative program analysis and deep neural networks. This will include exploring the effectiveness of CodeTrek in other domains and applications, including but not limited to smart contract security. Additionally, I will investigate ways to improve the scalability and efficiency of large-scale program analysis using CodeTrek.

6.1. Improving Smart Contract Security

The CodeTrek framework can help identify security bugs in smart contracts. It enables us to identify patterns that are indicative of common security vulnerabilities. Here are a few ways CodeTrek can assist us in finding smart contract security issues:

Vulnerability Detection. CodeTrek can be trained to recognize patterns in source code that are indicative of common yet severe security vulnerabilities, such as re-entrancy and unauthorized access. The models can then be used to scan complicated smart contracts and flag potential vulnerabilities.

Anomaly Detection. Blockchain transaction history and smart contract source code can be used to train CodeTrek models to detect patterns of behavior that deviate from expected behaviors. Using these models, one can detect suspicious activity in smart contracts, such as unexpected access to sensitive data or unexpected fund transfers.

Discovering Unexpected Behavior. The intended behavior of smart contracts is often specified in comments or documentation in the form of plain English. NLP techniques can be used to extract information from these comments and documentation. This information can be used along with the CodeTrek framework to train models that can detect bugs and

unexpected behavior with respect to the provided specification.

Code Review. Human code reviews are expensive and time-consuming. To alleviate this cost, CodeTrek can be trained to understand the complicated semantics of smart contracts to automatically flag potential bugs or vulnerabilities for human review.

Integration with other Tools. By integrating CodeTrek with tools other than Semmle, its capabilities can be extended. For example, we can use MAIAN (Nikolić et al., 2018) to create a labeled dataset for finding contracts that lock funds indefinitely. This tool—like any other dynamic analyzer—is expensive to operate. It generates long sequences of invocations of a contract which can be used as an auxiliary input to Codetrek at training time.

6.2. Improving the Scalability of Program Analysis Tools

As software systems become larger and more complex, there is a growing need for techniques that can handle large codebases. Future work on CodeTrek could involve using neural models to analyze large codebases in a more efficient and scalable way. One advantage of CodeTrek is that it relies on random walks, which are computationally efficient, making it well-suited to large systems. Some types of program analyses, such as those that use global dataflow analysis, can be expensive. By integrating a CodeTrek model into the logic query that calculates dataflow edges, this cost can be reduced. For instance, the logic query can calculate only local dataflow edges and when a global dataflow edge is needed, the CodeTrek model is called.

6.3. Applying the Framework to Different Applications

This dissertation only used CodeTrek for detecting a limited set of programming bugs and producing short summaries of code. It may be possible to extend the benefits of this framework to other applications.

Code Generation. Code generation models that are at the forefront of technology have been proven to be very accurate and scalable. The models do, however, have some limitations, including the lack of reasoning ability and explainability, as well as the need for large amounts of labeled data. CodeTrek is a semantic-heavy framework that may address

these issues and generate more complex and secure code, such as code for web applications or distributed systems.

Software Development. The CodeTrek framework may also be useful for developing models that identify bugs at various stages of the software development lifecycle. For example, CodeTrek models can be trained to detect discrepancies in design documents or use case specifications, as code is being written. This facilitates the detection of bugs early in the development process. In addition, models can be trained to detect bugs during the testing phase, by analyzing test cases.

Performance Issues. Last but not least, CodeTrek can be used for developing models that can detect more types of bugs, such as performance issues, by incorporating available dynamic information in its relational graph. For example, detecting performance issues can be done by training models to recognize patterns in the source code that are indicative of performance bottlenecks. These patterns can include inefficient algorithms or excessive memory usage.

CHAPTER 7

Related Work

7.1. Learning to Represent Code

There is a rich literature on using neural networks for code reasoning. At the token sequence level, the Transformer and its variants (Hellendoorn et al., 2020; Dowdell and Zhang, 2020) have been widely used (Berabi et al., 2021; Ahmad et al., 2020; Zügner et al., 2021; Kim et al., 2021; Wang et al., 2020). Their performance can be further boosted via pretraining (Feng et al., 2020; Guo et al., 2020a; Kanade et al., 2020; Wang et al., 2021; Peng et al., 2021; Liu et al., 2020; Li et al., 2022; Chen et al., 2021). Others have proposed to represent programs with ASTs and additional semantic edges (Allamanis et al., 2018; Brockschmidt et al., 2018; Guo et al., 2020a) or learned abstract relations (Johnson et al., 2020), using GNN or leaf-to-leaf sequence embeddings (Alon et al., 2018, 2019b). In this dissertation, I present a new approach for representing programs that enables adding rich semantic information. Additionally, the proposed approach takes advantage of program analysis queries on relational databases to eliminate the engineering burden of augmenting program graphs with additional semantic edges.

7.2. Graph Representation Learning

My work on learning program representations via relational databases is closely related to inductive representation learning on graphs (Hamilton et al., 2017) with graph neural networks (Xu et al., 2018) or Transformers (Ying et al., 2021). Although scalable GNNs via sampling (Chen et al., 2017; Zhou et al., 2020) have been proposed in the transductive setting, it is still challenging to represent large database graphs with 100k nodes in this inductive setting (Clement et al., 2021; Yang and Kuang, 2021). Techniques from transductive graph embedding based on skip-gram (Perozzi et al., 2014; Grover and Leskovec, 2016) or general knowledge graph embedding (Das et al., 2018; Hamilton et al., 2018; Zheng et al., 2020) are scalable but not directly applicable for inductive setting. In this dissertation, I

present an alternative approach that achieves a good balance between modeling for large codebases and efficiency.

7.3. Automated Feature Selection and Augmentation

Feature selection usually refers to reducing a large set of features to a smaller set of useful features. This is especially true when analyzing large data lakes with a high degree of diversity (Chepurko et al., 2020). There are efforts in the data-mining literature to minimize human effort in feature augmentation and selection. Chepurko et al. (2020) discover joins that can improve the prediction accuracy for a *single* data table. A similar approach (Yakout et al., 2012) automatically searches the the web to find relevant information in order to augment the user-provided data. The techniques presented in this dissertation are orthogonal to these works.

7.4. Sampling Large Graphs

Large graphs have been examined in several studies. Some of them perform various sampling schemas such as mini-batching (Zeng et al., 2020; Kipf and Welling, 2016; Hamilton et al., 2017; Chen et al., 2018) or attention mechanism (Veličković et al., 2018) to enable training on giant graphs. Various efforts have also been made to improve the efficiency of managing and training large graphs, including (Xie et al., 2021a; Gandhi and Iyer, 2021; Zhu et al., 2019; Chiang et al., 2019). While these approaches are typically used for recommendation tasks that involve one huge graph, the technique I propose takes into account many program graphs when tackling code understanding problems.

7.5. Walk-based Embedding

A number of representation learning techniques on graphs use random walks to learn node embeddings. DeepWalk (Perozzi et al., 2014) uses simple unbiased random walks to measure node similarity. node2vec (Grover and Leskovec, 2016) builds on DeepWalk by introducing two hyperparameters to control the tendency for random walks to traverse more depth or breadth. Further extensions include Walklets (Perozzi et al., 2017), which skip over steps in random walks to allow short walks to capture multiple levels of relationships, as well

as neural embedding approaches that employ hyperbolic rather than Euclidean spaces to define node similarities (Chamberlain et al., 2017). Unlike these approaches, my proposed technique aims to learn and guide traversal of graphs, rather than depending on simple unbiased random walks.

7.6. Representation Learning over Relational Databases

Deep learning models can be used for embedding structured tabular data into a latent space (Arora et al., 2021). Arora et al. (2021) presents an attention-based model learns node embeddings by capturing the semantic relationships across tables. Table2Vec (Zhang et al., 2019) assumes each row to be a sequence of tokens, and trains a model over these sequences. EmbDi (Cappuzzo et al., 2021) captures the semantics of entities by connecting it to the corresponding row identifier. They sample random walks over each node and embed the walks as token sequences. Finally, RelBERT (Arora et al., 2021) introduces an attention-based model based on the table and column that each entity appears in. In contrast, the technique presented in this dissertation takes advantage of the schema of the relational database to construct a graph in which each row is a node, and each key-foreign-key relationship is an edge. Then, during the representation learning, guided walks are sampled from the graph and embedded using the contents of nodes, edges, and their attributes.

7.7. Automated Code Summarization

There are two general techniques for automated code summarization: information retrieval and deep learning (Cheng et al., 2022). Early studies (Haiduc et al., 2010; Sridhara et al., 2010) utilized information retrieval techniques. The idea was to extract the most pertinent keywords from the given input and generate an extractive summary using them. The problem with these *extractive* summarization methods is that they are often inflexible and unable to generate coherent natural language descriptions (Cheng et al., 2022). Several recent studies have used deep learning to improve *abstract* code summarization. It has greatly benefited from advances in Neural Machine Translation (NMT). For learning a mapping between two languages, NMT generally uses the encoder-decoder model. In NMT-based code

summarization techniques, the encoder-decoder architecture is used to map code snippets to their natural language descriptions (Iyer et al., 2016). Nevertheless, other researchers argue that the characterization of code snippets requires more than token sequences, e.g., the structure information extracted from Abstract Syntax Trees (Hu et al., 2018a, 2020; LeClair et al., 2019; Alon et al., 2018; Wan et al., 2018). Recent works (Ahmad et al., 2020) take advantage of the Transformer architecture (Vaswani et al., 2017) as their base encoder-decoder framework. They use token sequences of codes as model inputs. Several other works (Gao and Lyu, 2022; Yang et al., 2021; Zhang et al., 2020b) combine token sequences with structural information to improve code summarization.

7.8. Robust Deep Neural Networks

Most deep neural models of code are not robust. The lack of robustness has been studies from two perspectives (Bui and Yu, 2022): 1) adversarial robustness, and 2) out-of-distribution data. The former shows that small perturbations to the input lead to incorrect predictions (Bielik and Vechev, 2020; Henke et al., 2022; Rabin et al., 2021; Yefet et al., 2020; Zhang et al., 2020a; Gao et al., 2020). These perturbations are often in the form of semantics-preserving transformations. The latter shows that models of code fail to produce correct results when they see an input that differs from the training data (Bui and Yu, 2022).

There has been ample effort on making models of code robust against adversarial attacks (Bielik and Vechev, 2020). For example, Henke et al. (2022) shows how to perform adversarial training to build models that are robust against semantics-preserving transformations. Yefet et al. (2020) explores various defense techniques against adversarial examples. Some of these techniques require re-training (possibly using a modified loss function or a modified version of the original training set) whereas the rest can be plugged in on top of existing trained models. An example of a conservative defensive approach is replacing all variables to an UNK symbol at test or training time. However, there are fewer studies on tackling out-of-distribution input. One of the most recent solutions for this challenge proposes to enable source code models to say "I don't know" whenever possible instead of

Authors	Definition
(Kim et al., 2016)	The degree to which a human can consistently predict the model's result.
(Doshi-Velez and Kim, 2017)	Ability to explain or to present in understandable terms to a human.
(Miller, 2019)	The degree to which a human can understand the cause of a decision.
(Molnar, 2020)	Methods and models that make the behavior and predictions of machine learning systems understandable to humans.

Table 7.1: Interpretability Definitions.

making a blind prediction by pretending that they knew the answer (Bui and Yu, 2022).

7.9. Interpretable Deep Neural Networks

There has been multiple attempts at defining interpretability in machine learning over the past years. All these definitions revolve around the strength of humans in intuition and reasoning. These definitions are illustrated in Table 7.1. According to Miller (2019), the interpretability of a model is higher if it is easier for a person to reason and trace back why a prediction was made by the model. Thus, interpretability is clearly related to humans' ability to grasp information through observation and reasoning (Carvalho et al., 2019).

There are two major paths towards interpretability: 1. creating intrinsically interpretable models and 2. creating explanation methods which are applicable to existing blackbox models (Carvalho et al., 2019).

CHAPTER 8

Conclusion

In this dissertation, I proposed to model programs as relational databases and demonstrated the power of sampling biased random walks over relational graphs in tackling three major challenges in the AI4Code community: (1) representing code in a systematic and extensible way, (2) maintaining the robustness of models when faced with out-of-distribution input, and (3) being able to explain why a model makes a particular prediction in a human-understandable way. I demonstrated the superiority of relational representation of code in terms of performance, robustness, and interpretability through several qualitative and quantitative evaluations.

GLOSSARY

Throughout the dissertation, several terms are used but not directly defined. This glossary includes their definitions.

Accuracy. The percentage of examples for which the model produces correct results.

Adam. *or* Adaptive Moment Estimation. A adaptive learning rate optimization algorithm based on exponentially decaying averages of past gradients and squared gradients.

Attention Pooling. Attention pooling behaves like a normal self-attention mechanism with a few adjustments to transform attention into a learnable pooling mechanism. Attention pooling uses the self-attention mechanism to choose the right set of two-dimensional features to transform into one-dimensional features.

Cross-Entropy Loss. A function that measures the difference between the empirical distribution defined by the training set and the probability distribution defined by the model.

E-M Algorithm. *or* Expectation-Maximization Algorithm. E-M is a training algorithm for models with latent variables. There are two steps in the algorithm: (E-step) Find the expected value of the log-likelihood on the observed data with current estimates of the parameters. (M-step) Update parameter estimates to increase the likelihood. These two steps are alternated until convergence is achieved.

Foreign Key. In relational databases, a foreign key is a column or set of columns that links data in two tables. By referencing the primary key of another table, it establishes a link between them.

k-fold Cross-Validation. an algorithm to estimate the generalization error of a learning algorithm when the given dataset is too small for a simple train/test or train/valid split to yield accurate estimation of generalization error because the mean of a loss on a small test set may have too high a variance.

Mean Pooling. This operation calculates the average value for patches of a feature map, and uses it to create a downsampled (pooled) version.

MLP. *or* multilayer perceptron. Also known as a feedforward deep network, a MLP is a mathematical function mapping some set of input values to output values.

Primary Key. In relational databases, a primary key is a column or combination of columns that uniquely identifies each row.

Query. A query is a question that a user formulates to obtain information from a database.

Query Language. The language through which users interact with database systems. It defines data structures in the database, and allows fast retrieval and modification of data.

Referential Integrity Constraint. Referential integrity requires that foreign keys have a corresponding primary key, or they must be null. This constraint maintains the correspondence between rows in two tables.

Relation. Each relation (or table) in a relational database contains one or more data categories in columns or attributes.

Relational Database. A relational database is a repository of information that organizes data in established relationships.

Relationship. The interactions between entities in a database.

ROC Curve. *or* Receiver Operating Characteristics Curve. ROC curve plots the true positive rate (i.e., recall) against the false positive rate (i.e., FPR). To compute the ROC curve, one has to compute the recall and FPR for various threshold values. One way to compare classifiers is to measure the area under the curve (AUC). A perfect classifier has a ROC-AUC of 1.0, whereas a purely random classifier has a ROC-AUC of 0.5.

SGD. *or* Stochastic Gradient Descent. The most commonly applied optimization algorithms for machine learning in general and for deep learning in particular. At every training iteration, SGD samples a minibatch of m samples from the training set. It then computes a gradient estimate by taking the average gradient on a minibatch of those samples. It applies the updates at the end of each iteration. SGD handles large datasets efficiently because it operates on training instances independently.

Sub-tokenization. *or* Subword Tokenization. Subword tokenization is a strategy from machine translation that breaks words into "subword units"—strings of characters like "ing" or "eau"—that allows the downstream model to make intelligent decisions about words it does not recognize. This strategy greatly reduces the size of the vocabulary.

WordPiece. A subword tokenization technique. WordPiece pre-tokenizes text into words (by splitting punctuation and whitespace) and tokenizes each word into subwords, called wordpieces. WordPiece tokenizes a single word using a greedy longest-match-first strategy, that is, it iteratively selects the longest prefix of the remaining text that matches a word in the model's vocabulary.

APPENDIX A

Full Task Specification Syntax

This section describes the full task specification syntax that is introduced in Section 3.3.

A.1. Walk and Task Specifications

In Table A.1, walk and task specification syntax can be found. All the tokens in **bold** font are keywords. Commas (**,**), curly braces (**{}**), colons (**:**), and semi-colons (**;**) are also part of the language. Any token in *italics* is a non-terminal. INT is any integer literal. If no score is specified for a relation name, then its score is initialized to 1. "relationName" is any available relation name. For a full list of available Python relation names refer to Appendix B.

walk_spec ::=	**{**	*task_spec* ::=	**{**
C =	*predicate* **,**	**RB** =	*relations* **,**
B =	*scores* **,**	**RQ** =	*relations* **,**
min =	INT **,**	**S** =	*walk_spec* **,**
max =	INT	**N** =	INT
}		**}**	
scores ::=	**{** *score_tuples* **}**	*relations* ::=	**{** *rels* **}**
score_tuples ::=	*score* | *score_tuples*	*rels* ::=	*rel* | *rels*
score ::=	*rel* **:** INT **;**	*rel* ::=	relationName

Table A.1: Walk and Task Specification.

A.2. Predicates and Formulas

Table A.2 illustrates the full syntax of predicates and formulas. The syntax is similar to that of CodeQL. All the tokens in **bold** font are keywords. The keyword **x** refers to any node, and the formula that comes after the colon (**:**) in front of it is used to filter specific nodes. Moreover, **random** is a keyword that can be used when no particular conditions are defined on anchors and they should be sampled randomly.

predicate ::=	**{ x :** *formula* **}** | **random**
formula ::=	*fparen* | *disjunction* | *conjunction* | *negation* | *instanceof* | *isliteral* | *call*
fparen ::=	**(** *formula* **)**
disjunction ::=	*formula* **or** *formula*
conjunction ::=	*formula* **and** *formula*
call ::=	*predicateRef* **(** (*exprs*)? **)** | *primary* **.** *predicateName* **(** (*exprs*)? **)**
negation ::=	**not** *formula*
instanceof ::=	*expr* **instanceof** *type*
isliteral ::=	*expr* **is** *literal*

Table A.2: Syntax of Predicates and Formulas.

A.3. Expressions

Table A.3 illustrates non-terminals that are made up of identifiers. "lowerId" is an identifier that starts with a lower-case letter. "upperId" is an identifier that starts with an upper-case letter. Anything between a pair of parentheses and a question mark at the front (e.g., "(*exprs*)?") refers to an occurrence of zero or one.

simpleId ::=	lowerId | upperId
classname ::=	upperId
variable ::=	lowerId
predicateName ::=	lowerId
moduleId ::=	*simpleId* | *moduleId* **::** *simpleId*
predicateRef ::=	(*moduleId* **::**)? lowerId

Table A.3: Identifiers.

Table A.4 illustrates the full syntax of expressions. The syntax is similar to that of CodeQL. All the tokens in **bold** font are keywords. All the tokens in *italic* are non-terminals. INT, FLOAT, and STRING are integer, floating point, and string literals, respectively.

expr ::=	*unop* | *binop* | *primary*
primary ::=	*eparen* | *literal* | *variable* | *call_res* | *range*
eparen ::=	**(** *expr* **)**
literal ::=	**false** | **true** | INT | FLOAT | STRING
unop ::=	+ *expr* | **-** *expr*
binop ::=	*expr* + *expr* | *expr* **-** *expr* | *expr* ***** *expr* | *expr* / *expr* | *expr* % *expr*
call_res ::=	*predicateRef* **(** (*expr*)? **)** | *primary* **.** *predicateName* **(** (*expr*)? **)**
range ::=	**[** *expr* **..** *expr* **]**

Table A.4: Syntax of Expressions.

A.4. Types

Since QL is a statically typed language, each variable has a declared type. In Table A.5, boolean, float, int, and string are primitive QL types and are language keywords. **boolean** contains **true** and **false** values, **float** contains 64-bit floating point numbers, **int** contains 32-bit two's complement integers, and **string** contains finite strings of 16-bit characters.

type ::=	(*moduleId* **::**)? *classname* | **boolean** | **float** | **int** | **string**

Table A.5: Types.

APPENDIX B

List of Python Relations

This section contains a full list of available relation names for Python.

Table B.1: Python Base Relations in CodeQL

locations_ast	py_false_successors	py_ssa_phi
py_boolops	py_flow_bb_node	py_ssa_use
py_bytes	py_Functions	py_ssa_var
py_Classes	py_idoms	py_stmt_lists
py_cmpop_lists	py_ints	py_stmts
py_cmpops	py_locations	py_str_lists
py_comprehension_lists	py_Modules	py_strs
py_comprehensions	py_numbers	py_successors
py_dict_item_lists	py_operators	py_true_successors
py_dict_items	py_parameter_lists	py_unaryops
py_exception_successors	py_scope_flow	py_variables
py_expr_contexts	py_scope_location	variable
py_expr_lists	py_scopes	py_cobjectnames
py_exprs	py_ssa_defn	py_cobjects
py_cobjecttypes	py_decorated_object	py_line_lengths
py_codelines	py_docstringlines	py_module_path
py_commentblocks	py_exports	py_special_objects
py_commentlines	py_extracted_version	py_StringPart_lists
py_comments	py_flags_versioned	py_StringParts
containerparent	files	folders
locations_default	numlines	py_absolute_names
py_alias_lists	py_aliases	py_alllines
py_arguments	py_bools	py_citems
py_cmembers_versioned	tokens	py_cobject_sources

Table B.2: Python Derived Relations in CodeQL

AlertSuppression	ContainsNonContainer
AssertLiteralConstant	ContextEfficiency
AssertOnTuple	ContextMarginalEfficiency
BackspaceEscape	CookieInjection
BadTagFilter	CSRFProtectionDisabled
BindToAllInterfaces	CsvInjection
BreakOrReturnInFinally	CyclicImport
BrokenCryptoAlgorithm	CyclomaticComplexity
C_StyleParentheses	DefineEqualsWhenAddingAttributes
CallableDisplayStrings	Definitions
CallableExtents	DeprecatedModule
CallableSourceLinks	DeprecatedSliceMethod
CallGraph	DirectImports
CallGraphEfficiency	DocStringRatio
CallGraphMarginalEfficiency	DocStrings
CallToSuperWrongClass	DuplicateBlock
CatchingBaseException	DuplicateCharacterInSet
ClassAfferentCoupling	DuplicateFunction
ClassDisplayStrings	DuplicateKeyInDictionaryLiteral
ClassEfferentCoupling	Efficiency
ClassExtents	EmptyExcept
ClassifyFiles	EncodingError
ClassSourceLinks	EqualsNone
CleartextLogging	EqualsOrHash
CleartextStorage	EqualsOrNotEquals
ClientSuppliedIpUsedInSecurityCheck	ExecUsed
CLinesOfCode	ExpectedMappingForFormatString
CodeInjection	ExplicitCallToDel
CommandInjection	ExplicitReturnInInit
CommentedOutCode	ExternalAPIsUsedWithUntrustedData
CommentRatio	ExternalDependencies
CommitDisplayStrings	ExternalDependenciesSourceLinks
CommitSourceLinks	ExtractionWarnings
CompareConstants	FailedInference
CompareIdenticalValues	FClasses
CompareIdenticalValuesMissingSelf	FCommentedOutCode
ConflictingAttributesInBaseClasses	FFunctionsAndMethods
Consistency	FileNotAlwaysClosed
ConsistentReturns	FlaskDebug
ConstantInConditional	FLines

Table B.3: Python Derived Relations in CodeQL (continued)

FLinesOfCode	IncorrectlySpecifiedOverriddenMethod
FLinesOfComments	IncorrectRaiseInSpecialMethod
FLinesOfDuplicatedCode	InitCallsSubclassMethod
FLinesOfSimilarCode	InitIsGenerator
FNumberOfTests	InsecureCookie
FromImportOfMutableAttribute	InsecureDefaultProtocol
FullServerSideRequestForgery	InsecureProtocol
FunctionNumberOfCalls	InsecureRandomness
FunctionStatementNestingDepth	InsecureTemporaryFile
Global	IterableStringOrSequence
GlobalAtModuleLevel	IterReturnsNonIterator
HardcodedCredentials	IterReturnsNonSelf
HashedButNoHash	Jinja2WithoutEscaping
HChurn	JWTEmptyKeyOrAlgorithm
HeaderInjection	JWTMissingSecretOrPublicKeyVerification
HLinesAdded	KeyPointsToFailure
HLinesDeleted	LackofCohesionInMethodsCK
HNumberOfAuthors	LackofCohesionInMethodsHM
HNumberOfCoCommits	LdapInjection
HNumberOfCommits	LDAPInsecureAuth
HNumberOfRecentAuthors	LeakingListComprehension
HNumberOfRecentChangedFiles	LinesOfCode
HNumberOfRecentCommits	LinesOfUserCode
HNumberOfReCommits	LocalDefinitions
IllegalExceptionHandlerType	LocalReferences
IllegalRaise	LogInjection
ImportandImportFrom	LoopVariableCapture
ImportFailure	MaybeUndefinedClassAttribute
Imports	MismatchInMultipleAssignment
ImportShadowedByLoopVar	MissingCallToDel
ImportStarUsed	MissingCallToInit
ImpreciseAssert	MissingHostKeyValidation
ImproperLdapAuth	MissingPartSpecialGroup
IncompleteHostnameRegExp	MixedExplicitImplicitIn3101Format
IncompleteOrdering	ModificationOfLocals
IncompleteUrlSubstringSanitization	ModificationOfParameterWithDefault
InconsistentMRO	ModuleAfferentCoupling
IncorrectComparisonUsingIs	ModuleEfferentCoupling
IncorrectExceptOrder	ModuleImportsItself
IncorrectlyOverriddenMethod	ModuleLevelCyclicImport

Table B.4: Python Derived Relations in CodeQL (continued)

MostlyDuplicateClass	RemoteFlowSources
MostlyDuplicateFile	RemoteFlowSourcesReach
MostlySimilarFile	RequestHandlers
MultipleImports	RequestWithoutValidation
MultiplyDefined	ResolvableCallCandidates
MutatingDescriptor	ReturnConsistentTupleSizes
NamingConventionsClasses	ReturnOrYieldOutsideFunction
NamingConventionsFunctions	ReturnValueIgnored
NestedLoopsSameVariable	ShadowBuiltin
NestedLoopsSameVariableWithReuse	ShadowGlobal
NonCallableCalled	ShouldBeContextManager
NonCls	ShouldUseWithStatement
NonIteratorInForLoop	SideEffectInAssert
NonPortableComparisonUsingIs	SignatureOverriddenMethod
NonSelf	SignatureSpecialMethods
NoSQLInjection	SimilarFunction
NotImplementedIsNotAnException	SimpleXmlRpcServer
NumberOfParametersWithoutDefault	SlotsInOldStyleClass
NumberOfStatements	SqlInjection
OldOctalLiteral	StackTraceExposure
OverlyComplexDelMethod	StatementNoEffect
OverwritingAttributeInSuperClass	StringConcatenationInLoop
PamAuthorization	SubclassShadowing
PartialServerSideRequestForgery	SuccessfullyExtractedFiles
PathInjection	Summary
PointsToFailure	SuperclassDelCalledMultipleTimes
PointsToResolvableCallRatio	SuperclassInitCalledMultipleTimes
PointsToResolvableCalls	SuperInOldStyleClass
PointsToResolvableCallsRelevantTarget	SuspiciousUnusedLoopIterationVariable
PolynomialReDoS	SyntaxError
PropertyInOldStyleClass	TarSlip
Pruned	TemplateInjection
Pythagorean	ToDoComment
RaisingTuple	TopLevelPrint
RatioOfDefinitions	TransitiveImports
ReDoS	TruncatedDivision
RedundantAssignment	TypeHierarchyFailure
ReflectedXSS	TypeInferenceFailure
ReflectedXss	UndefinedClassAttribute
RegexInjection	UndefinedExport

Table B.5: Python Derived Relations in CodeQL (continued)

UndefinedGlobal	UrlRedirect
UndefinedPlaceHolder	UseImplicitNoneReturnValue
UnguardedNextInGenerator	UselessClass
UninitializedLocal	UselessComparisonTest
UnintentionalImplicitStringConcatenation	UseofApply
UnintentionalImport	UseOfExit
UnmatchableCaret	UseofInput
UnmatchableDollar	WeakCryptoKey
UnnecessaryDelete	WeakFilePermissions
UnnecessaryElseClause	WeakSensitiveDataHashing
UnnecessaryLambda	WrongNameForArgumentInCall
UnnecessaryPass	XmlBomb
UnreachableCode	XpathInjection
UnsafeDeserialization	Xslt
UnsupportedFormatCharacter	Xxe
UntrustedDataToExternalAPI	ZipSlip
UnusedArgumentIn3101Format	
UnusedExceptionObject	
UnusedImport	
UnusedLocalVariable	
UnusedModuleVariable	
UnusedNamedArgumentIn3101Format	
UnusedParameter	
WrongNumberArgumentsInCall	
WrongNumberArgumentsForFormat	
WrongNumberArgumentsFor3101Format	
WrongNameInArgumentsFor3101Format	
WrongNameForArgumentInClassInstantiation	
WrongNumberArgumentsInClassInstantiation	

APPENDIX C

CodeQL Queries used for Labeling

In this section, I present the CodeQL queries that we used to label the examples for newly added tasks. Both queries are adapted from the official CodeQL's query repository at https://github.com/github/codeql.

C.1. DefUse-Fun Query

```
import python
import Definition

predicate unused_local(Name unused, LocalVariable v) {
  forex(Definition def | def.getNode() = unused |
    def.getVariable() = v
    and def.isUnused()
    and not exists(def.getARedef())
    and not exists(ann_wo_assignment(v))
    and def.isRelevant()
    and not v = any(Nonlocal n).getAVariable()
    and not exists(def.getNode().getParentNode()
                        .(FunctionDef).getDefinedFunction()
                        .getADecorator())
    and not exists(def.getNode().getParentNode()
                        .(ClassDef).getDefinedClass()
                        .getADecorator())
  )
}

private AnnAssign ann_wo_assignment(LocalVariable v) {
```

```
22    result.getTarget() = v.getAStore()
23    and not exists(result.getValue())
24  }
25
26  from Name unused, LocalVariable v
27  where
28    unused_local(unused, v) and
29    forall(Name el | el = unused.getParentNode().(Tuple).
       getAnElt()
30                   | unused_local(el, _))
31  select unused, v.getId()
```

C.2. VarShadow Query

```
1  import python
2  import semmle.python.types.Builtins
3
4  predicate optimizing_parameter(Parameter p) {
5    exists(string name, Name glob | p.getDefault() = glob
6                                  | glob.getId() = name
7      and p.asName().getId() = name
8    )
9  }
10
11 predicate shadows(Name d, GlobalVariable g,
12                   Function scope, int line) {
13   g.getScope() = scope.getScope()
14   and d.getScope() = scope
15   and exists(LocalVariable l |
16     d.defines(l) and
```

```
17      l.getId() = g.getId()
18    )
19    and not exists(Import il, Import ig, Name gd |
20                    il.contains(d)
21                    and gd.defines(g)
22                    and ig.contains(gd))
23    and not exists(Assign a | a.getATarget() = d
24    and a.getValue() = g.getAnAccess())
25    and not exists(Builtin::builtin(g.getId()))
26    and d.getLocation().getStartLine() = line
27    and exists(Name defn | defn.defines(g)
28                         | not exists(If i | i.isNameEqMain()
29                         | i.contains(defn)))
30    and not optimizing_parameter(d)
31  }
32
33  AttrNode pytest_fixture_attr() {
34    exists(ModuleValue pytest
35              | result.getObject("fixture").pointsTo(pytest))
36  }
37
38  Value pytest_fixture() {
39    exists(CallNode call |
40      call.getFunction() = pytest_fixture_attr()
41      or call.getFunction().(CallNode).getFunction() =
     pytest_fixture_attr()
42    | call.pointsTo(result)
43    )
```

```
}

predicate assigned_pytest_fixture(GlobalVariable v) {
  exists(NameNode def |
    def.defines(v) and def.(DefinitionNode).getValue()
                           .pointsTo(pytest_fixture())
  )
}

predicate first_shadowing_def(Name d, GlobalVariable g) {
  exists(int first, Scope scope |
    shadows(d, g, scope, first)
    and first = min(int line | shadows(_, g, scope, line))
  )
}

from Name d, GlobalVariable g, Name def
where
  first_shadowing_def(d, g)
  and not exists(Name n | n.deletes(g))
  and def.defines(g)
  and not assigned_pytest_fixture(g)
  and not g.getId() = "_"
select d, g.getId(), def
```

BIBLIOGRAPHY

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. A transformer-based approach for source code summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL)*, 2020.

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=BJOFETxR-.

Miltiadis Allamanis, Henry Jackson-Flux, and Marc Brockschmidt. Self-supervised bug detection and repair. *arXiv preprint arXiv:2105.12787*, 2021.

Uri Alon and Eran Yahav. On the bottleneck of graph neural networks and its practical implications. *arXiv preprint arXiv:2006.05205*, 2020.

Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400*, 2018.

Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. Structural language models for any-code generation. 2019a.

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL): 1–29, 2019b.

Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. Structural language models of code. In *International Conference on Machine Learning*, pages 245–256. PMLR, 2020.

Siddhant Arora, Vinayak Gupta, Garima Gaur, and Srikanta Bedathur. Bert meets relational db: Contextual representations of relational databases. *arXiv preprint arXiv:2104.14914*, 2021.

Pavel Avgustinov, Oege De Moor, Michael Peyton Jones, and Max Schäfer. Ql: Object-oriented queries on relational data. In *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

Satanjeev Banerjee and Alon Lavie. Meteor: An automatic metric for mt evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, pages 65–72, 2005.

Berkay Berabi, Jingxuan He, Veselin Raychev, and Martin Vechev. Tfix: Learning to fix coding errors with a text-to-text transformer. In *International Conference on Machine Learning*, pages 780–791. PMLR, 2021.

Pavol Bielik and Martin Vechev. Adversarial robustness for code. In *International Conference on Machine Learning*, pages 896–907. PMLR, 2020.

Marc Brockschmidt, Miltiadis Allamanis, Alexander L Gaunt, and Oleksandr Polozov. Generative code modeling with graphs. *arXiv preprint arXiv:1805.08490*, 2018.

Nghi DQ Bui and Yijun Yu. Towards robust models of code via energy-based learning on auxiliary datasets. In *2022 IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2022.

Riccardo Cappuzzo, Paolo Papotti, and Saravanan Thirumuruganathan. Embdi: Generating embeddings for relational data integration. 2021.

Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *2017 ieee symposium on security and privacy (sp)*, pages 39–57. Ieee, 2017.

Diogo V Carvalho, Eduardo M Pereira, and Jaime S Cardoso. Machine learning interpretability: A survey on methods and metrics. *Electronics*, 8(8):832, 2019.

Benjamin Paul Chamberlain, James Clough, and Marc Peter Deisenroth. Neural embeddings of graphs in hyperbolic space. *arXiv preprint arXiv:1705.10359*, 2017.

Jianfei Chen, Jun Zhu, and Le Song. Stochastic training of graph convolutional networks with variance reduction. *arXiv preprint arXiv:1710.10568*, 2017.

Jie Chen, Tengfei Ma, and Cao Xiao. FastGCN: Fast learning with graph convolutional networks via importance sampling. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=rytstxWAW.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harri Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Zimin Chen and Martin Monperrus. A literature study of embeddings on source code. *arXiv preprint arXiv:1904.03061*, 2019.

Wuyan Cheng, Po Hu, Shaozhi Wei, and Ran Mo. Keyword-guided abstractive code summarization via incorporating structural and contextual information. *Information and Software Technology*, 150:106987, 2022.

Nadiia Chepurko, Ryan Marcus, Emanuel Zgraggen, Raul Castro Fernandez, Tim Kraska,

and David Karger. Arda: Automatic relational data augmentation for machine learning. *arXiv preprint arXiv:2003.09758*, 2020.

Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019.

Kenneth Ward Church. Word2vec. *Natural Language Engineering*, 23(1):155–162, 2017.

Colin B Clement, Shuai Lu, Xiaoyu Liu, Michele Tufano, Dawn Drain, Nan Duan, Neel Sundaresan, and Alexey Svyatkovskiy. Long-range modeling of source code files with ewash: Extended window access by syntax hierarchy. 2021.

Rajarshi Das, Shehzaad Dhuliawala, Manzil Zaheer, Luke Vilnis, Ishan Durugkar, Akshay Krishnamurthy, Alex Smola, and Andrew McCallum. Go for a walk and arrive at the answer: Reasoning over paths in knowledge bases using reinforcement learning. In *International Conference on Learning Representations*, 2018.

Oege De Moor, Mathieu Verbaere, Elnar Hajiyev, Pavel Avgustinov, Torbjorn Ekman, Neil Ongkingco, Damien Sereni, and Julian Tibble. Keynote address:. ql for source code analysis. In *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, pages 3–16. IEEE, 2007.

Arthur P Dempster, Nan M Laird, and Donald B Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society: Series B (Methodological)*, 39(1):1–22, 1977.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. Hoppity: Learning graph transformations to detect and fix bugs in programs. In *International Conference on Learning Representations (ICLR)*, 2020.

Finale Doshi-Velez and Been Kim. Towards a rigorous science of interpretable machine learning. *arXiv preprint arXiv:1702.08608*, 2017.

Thomas Dowdell and Hongyu Zhang. Language modelling for source code with transformer-xl. *arXiv preprint arXiv:2007.15813*, 2020.

Mengnan Du, Ninghao Liu, and Xia Hu. Techniques for interpretable machine learning. *Communications of the ACM*, 63(1):68–77, 2019.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

Swapnil Gandhi and Anand Padmanabha Iyer. P3: Distributed deep graph learning at scale. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, pages 551–568, 2021.

Xiang Gao, Ripon K Saha, Mukul R Prasad, and Abhik Roychoudhury. Fuzz testing based data augmentation to improve robustness of deep neural networks. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 1147–1158. IEEE, 2020.

Yuexiu Gao and Chen Lyu. M2ts: Multi-scale multi-modal approach based on transformer for source code summarization. *arXiv preprint arXiv:2203.09707*, 2022.

Siddhant Garg and Goutham Ramakrishnan. Bae: Bert-based adversarial examples for text classification. *arXiv preprint arXiv:2004.01970*, 2020.

Floris Geerts, Filip Mazowiecki, and Guillermo Perez. Let's agree to degree: Comparing graph convolutional networks in the message-passing framework. In *International Conference on Machine Learning*, pages 3640–3649. PMLR, 2021.

Leilani H Gilpin, David Bau, Ben Z Yuan, Ayesha Bajwa, Michael Specter, and Lalana Kagal. Explaining explanations: An overview of interpretability of machine learning. In *2018 IEEE 5th International Conference on data science and advanced analytics (DSAA)*, pages 80–89. IEEE, 2018.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864, 2016.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020a.

Qipeng Guo, Xipeng Qiu, Pengfei Liu, Xiangyang Xue, and Zheng Zhang. Multi-scale self-attention for text classification. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 7847–7854, 2020b.

Sonia Haiduc, Jairo Aponte, and Andrian Marcus. Supporting program comprehension with source code summarization. In *2010 acm/ieee 32nd international conference on software engineering*, volume 2, pages 223–226. IEEE, 2010.

Patrick Hall, Navdeep Gill, Megan Kurka, and Wen Phan. Machine learning interpretability with h2o driverless ai. *H2O. ai*, 2017.

Will Hamilton, Payal Bajaj, Marinka Zitnik, Dan Jurafsky, and Jure Leskovec. Embedding logical queries on knowledge graphs. 31, 2018. URL https://proceedings.neurips.cc/paper/2018/file/ef50c335cca9f340bde656363ebd02fd-Paper.pdf.

William L Hamilton. Graph representation learning. *Synthesis Lectures on Artifical Intelligence and Machine Learning*, 14(3):1–159, 2020.

William L Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pages 1025–1035, 2017.

Jingxuan He, Luca Beurer-Kellner, and Martin Vechev. On distribution shift in learning-based bug detectors. *arXiv preprint arXiv:2204.10049*, 2022.

Vincent J Hellendoorn, Christian Bird, Earl T Barr, and Miltiadis Allamanis. Deep learning type inference. In *Proceedings of the 2018 26th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pages 152–162, 2018.

Vincent J Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. Global relational models of source code. In *International conference on learning representations*, 2020.

Jordan Henke, Goutham Ramakrishnan, Zi Wang, Aws Albarghouth, Somesh Jha, and Thomas Reps. Semantic robustness of models of source code. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 526–537. IEEE, 2022.

Sara Hooker, Dumitru Erhan, Pieter-Jan Kindermans, and Been Kim. A benchmark for interpretability methods in deep neural networks. *Advances in neural information processing systems*, 32, 2019.

Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pages 200–20010. IEEE, 2018a.

Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. Summarizing source code with transferred api knowledge. 2018b.

Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation with hybrid lexical and syntactical information. *Empirical Software Engineering*, 25(3):2179–2217, 2020.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2073–2083, 2016.

Daniel Johnson, Hugo Larochelle, and Daniel Tarlow. Learning graph structure with a finite-state automaton layer. 33:3082–3093, 2020. URL https://proceedings.neurips.cc/paper/2020/file/1fdc0ee9d95c71d73df82ac8f0721459-Paper.pdf.

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and evaluating contextual embedding of source code. In *International Conference on Machine Learning*, pages 5110–5121. PMLR, 2020.

Been Kim, Rajiv Khanna, and Oluwasanmi O Koyejo. Examples are not enough, learn to criticize! criticism for interpretability. *Advances in neural information processing systems*, 29, 2016.

Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. Code prediction by feeding trees to transformers. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 150–162. IEEE, 2021.

Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

Ching-Yun Ko, Zhaoyang Lyu, Lily Weng, Luca Daniel, Ngai Wong, and Dahua Lin. Popqorn: Quantifying robustness of recurrent neural networks. In *International Conference on Machine Learning*, pages 3468–3477. PMLR, 2019.

Vladimir Kovalenko, Egor Bogomolov, Timofey Bryksin, and Alberto Bacchelli. Pathminer: a library for mining of path-based representations of code. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 13–17. IEEE, 2019.

Triet HM Le, Hao Chen, and Muhammad Ali Babar. Deep learning for source code modeling and generation: Models, applications, and challenges. *ACM Computing Surveys (CSUR)*, 53(3):1–38, 2020.

Alexander LeClair, Siyuan Jiang, and Collin McMillan. A neural model for generating natural language summaries of program subroutines. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 795–806. IEEE, 2019.

Yi Li, Shaohua Wang, and Tien N Nguyen. Dlfix: Context-based code transformation learning for automated program repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 602–614, 2020.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814*, 2022.

Chin-Yew Lin. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, pages 74–81, 2004.

Chin-Yew Lin and Franz Josef Och. Orange: a method for evaluating automatic evaluation metrics for machine translation. In *COLING 2004: Proceedings of the 20th International Conference on Computational Linguistics*, pages 501–507, 2004.

Kang Lin and Yuzhuo Gao. Model interpretability of financial fraud detection by group shap. *Expert Systems with Applications*, 210:118354, 2022.

Zachary C Lipton. The mythos of model interpretability. *Queue*, 16(3):31–57, 2018.

Fang Liu, Ge Li, Yunfei Zhao, and Zhi Jin. Multi-task learning based pre-trained language model for code completion. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 473–485, 2020.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, MING GONG, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie LIU. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021. URL https://openreview.net/forum?id=6lE4dQXaUcb.

Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. *Advances in neural information processing systems*, 30, 2017.

Zheng Ma, Yuexiu Gao, Lei Lyu, and Chen Lyu. Mmf3: Neural code summarization based on multi-modal fine-grained feature fusion. In *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 171–182, 2022.

Tim Miller. Explanation in artificial intelligence: Insights from the social sciences. *Artificial intelligence*, 267:1–38, 2019.

Christoph Molnar. *Interpretable machine learning*. 2020.

Grégoire Montavon, Sebastian Lapuschkin, Alexander Binder, Wojciech Samek, and Klaus-Robert Müller. Explaining nonlinear classification decisions with deep taylor decomposition. *Pattern recognition*, 65:211–222, 2017.

Rohan Mukherjee, Yeming Wen, Dipak Chaudhari, Thomas Reps, Swarat Chaudhuri, and Christopher Jermaine. Neural program generation modulo static analysis. *Advances in Neural Information Processing Systems*, 34, 2021.

Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th annual computer security applications conference*, pages 653–663, 2018.

Cathy O'neil. *Weapons of math destruction: How big data increases inequality and threatens democracy*. Broadway books, 2016.

Lizi Ottens, Luis Perez, and Sudharshan Viswanathan. Automatic code generation using pre-trained language models.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.

Pardis Pashakhanloo, Aaditya Naik, Hanjun Dai, Petros Maniatis, and Mayur Naik. Learning to walk over relational graphs of source code. 2022a.

Pardis Pashakhanloo, Aaditya Naik, Yuepeng Wang, Hanjun Dai, Petros Maniatis, and Mayur Naik. Codetrek: Flexible modeling of code using an extensible relational representation. In *International Conference on Learning Representations (ICLR)*, 2022b.

Santanu Paul and Atul Prakash. Supporting queries on source code: A formal framework. *International Journal of Software Engineering and Knowledge Engineering*, 4(3):325–348, 1994.

Dinglan Peng, Shuxin Zheng, Yatao Li, Guolin Ke, Di He, and Tie-Yan Liu. How could neural networks understand programs? *arXiv preprint arXiv:2105.04297*, 2021.

Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710, 2014.

Bryan Perozzi, Vivek Kulkarni, Haochen Chen, and Steven Skiena. Don't walk, skip! online learning of multi-scale network embeddings. In *Proceedings of the 2017 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining 2017*, pages

258–265, 2017.

Michael Pradel and Koushik Sen. Deepbugs: A learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–25, 2018.

J Ross Quinlan. Decision trees as probabilistic classifiers. In *Proceedings of the Fourth International Workshop on Machine Learning*, pages 31–37. Elsevier, 1987.

Md Rafiqul Islam Rabin, Nghi DQ Bui, Ke Wang, Yijun Yu, Lingxiao Jiang, and Mohammad Amin Alipour. On the generalizability of neural program models with respect to semantic-preserving program transformations. *Information and Software Technology*, 135:106552, 2021.

Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. " why should i trust you?" explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1135–1144, 2016.

Devjeet Roy, Sarah Fakhoury, and Venera Arnaoudova. Reassessing automatic evaluation metrics for code summarization tasks. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1105–1116, 2021.

Yang Shi, Ye Mao, Tiffany Barnes, Min Chi, and Thomas W Price. More with less: Exploring how to use deep learning effectively through semi-supervised learning for automatic bug detection in student code. In *In Proceedings of the 14th International Conference on Educational Data Mining (EDM) 2021*, 2021.

Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. *arXiv preprint arXiv:1312.6034*, 2013.

Jing Kai Siow, Shangqing Liu, Xiaofei Xie, Guozhu Meng, and Yang Liu. Learning program semantics with code representations: An empirical study. *arXiv preprint arXiv:2203.11790*, 2022.

Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 43–52, 2010.

Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1433–1443, 2020.

Ajay Thampi. *Interpretable ai*. Simon and Schuster, 2022.

Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh Singh. Neural program repair by jointly learning to localize and repair. *arXiv preprint arXiv:1904.01720*, 2019.

Shobha Vasudevan, Wenjie Jiang, David Bieber, Rishabh Singh, HAMID SHOJAEI, C. Richard Ho, and Charles Sutton. Learning semantic representations to verify hardware designs. In A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, 2021. URL https://openreview.net/forum?id=oIhzg4GJeOf.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. In *International Conference on Learning Representations*, 2018.

Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S Yu. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, pages 397–407, 2018.

Deze Wang, Zhouyang Jia, Shanshan Li, Yue Yu, Yun Xiong, Wei Dong, and Xiangke Liao. Bridging pre-trained models and downstream tasks for source code understanding. In *Proceedings of the 44th International Conference on Software Engineering*, pages 287–298, 2022.

Wenhua Wang, Yuqun Zhang, Zhengran Zeng, and Guandong Xu. *trans*^{3}: A transformer-based framework for unifying code summarization and code search. *arXiv preprint arXiv:2003.03238*, 2020.

Xin Wang, Yasheng Wang, Pingyi Zhou, Meng Xiao, Yadao Wang, Li Li, Xiao Liu, Hao Wu, Jin Liu, and Xin Jiang. Clsebert: Contrastive learning for syntax enhanced code pre-trained model. *arXiv preprint arXiv:2108.04556*, 2021.

Bolin Wei, Yongmin Li, Ge Li, Xin Xia, and Zhi Jin. Retrieve and refine: exemplar-based neural comment generation. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 349–360. IEEE, 2020.

Max Welling and Thomas N Kipf. Semi-supervised classification with graph convolutional networks. In *J. International Conference on Learning Representations (ICLR 2017)*, 2016.

Anze Xie, Anders Carlsson, Jason Mohoney, Roger Waleffe, Shanan Peters, Theodoros Rekatsinas, and Shivaram Venkataraman. Demo of marius: a system for large-scale graph embeddings. *Proceedings of the VLDB Endowment*, 14(12):2759–2762, 2021a.

Rui Xie, Wei Ye, Jinan Sun, and Shikun Zhang. Exploiting method names to improve code summarization: A deliberation multi-task learning approach. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pages 138–148. IEEE, 2021b.

Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pages 1–10, 2022.

Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.

Mohamed Yakout, Kris Ganjam, Kaushik Chakrabarti, and Surajit Chaudhuri. Infogather: entity augmentation and attribute discovery by holistic matching with web tables. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 97–108, 2012.

Guang Yang, Xiang Chen, Jinxin Cao, Shuyuan Xu, Zhanqi Cui, Chi Yu, and Ke Liu. Comformer: Code comment generation via transformer and fusion method-based hybrid code representation. In *2021 8th International Conference on Dependable Systems and Their Applications (DSA)*, pages 30–41. IEEE, 2021.

Hao Yang and Li Kuang. Ccmc: Code completion with a memory mechanism and a copy mechanism. In *Evaluation and Assessment in Software Engineering*, pages 129–138. 2021.

Wuu Yang, Susan Horwitz, and Thomas Reps. A program integration algorithm that accommodates semantics-preserving transformations. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1(3):310–354, 1992.

Noam Yefet, Uri Alon, and Eran Yahav. Adversarial examples for models of code. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–30, 2020.

Chengxuan Ying, Tianle Cai, Shengjie Luo, Shuxin Zheng, Guolin Ke, Di He, Yanming Shen, and Tie-Yan Liu. Do transformers really perform bad for graph representation? *arXiv preprint arXiv:2106.05234*, 2021.

Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Russ R Salakhutdinov, and Alexander J Smola. Deep sets. *Advances in neural information processing systems*, 30, 2017.

Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna.

Graphsaint: Graph sampling based inductive learning method. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=BJe8pkHFwS.

Zhengran Zeng, Hanzhuo Tan, Haotian Zhang, Jing Li, Yuqun Zhang, and Lingming Zhang. An extensive study on pre-trained models for program understanding and generation. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 39–51, 2022.

Huangzhao Zhang, Zhuo Li, Ge Li, Lei Ma, Yang Liu, and Zhi Jin. Generating adversarial examples for holding robustness of source code processing models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 1169–1176, 2020a.

Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. Retrieval-based neural source code summarization. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 1385–1397. IEEE, 2020b.

Li Zhang, Shuo Zhang, and Krisztian Balog. Table2vec: Neural word and entity embeddings for table population and retrieval. In *Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 1029–1032, 2019.

Qirun Zhang, Chengnian Sun, and Zhendong Su. Skeletal program enumeration for rigorous compiler testing. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 347–361, 2017.

Yu Zhang, Peter Tiňo, Aleš Leonardis, and Ke Tang. A survey on neural network interpretability. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 2021.

Da Zheng, Xiang Song, Chao Ma, Zeyuan Tan, Zihao Ye, Jin Dong, Hao Xiong, Zheng Zhang, and George Karypis. Dgl-ke: Training knowledge graph embeddings at scale. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 739–748, 2020.

Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *AI Open*, 1:57–81, 2020.

Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. Aligraph: A comprehensive graph neural network platform. *Proc. VLDB Endow.*, 2019.

Daniel Zügner, Tobias Kirschstein, Michele Catasta, Jure Leskovec, and Stephan Günnemann. Language-agnostic representation learning of source code from structure and context. *arXiv preprint arXiv:2103.11318*, 2021.