CODETREK: FLEXIBLE MODELING OF CODE USING AN EXTENSIBLE RELATIONAL REPRESENTATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Designing a suitable representation for code-reasoning tasks is challenging in aspects such as the kinds of program information to model, how to combine them, and how much context to consider. We propose CodeTrek, a deep learning approach that addresses these challenges by representing codebases as databases that conform to rich relational schemas. The relational representation not only allows CodeTrek to uniformly represent diverse kinds of program information, but also to leverage program-analysis queries to derive new semantic relations, which can be readily incorporated without further architectural engineering. CodeTrek embeds this relational representation using a set of walks that can traverse different relations in an unconstrained fashion, and incorporates all relevant attributes along the way. We evaluate CodeTrek on four diverse and challenging Python tasks: variable misuse, exception prediction, unused definition, and variable shadowing. CodeTrek achieves an accuracy of 91%, 63%, 98%, and 94% on these tasks respectively, and outperforms state-of-the-art neural models by 2-19% points.

1 INTRODUCTION

Deep learning techniques are increasingly applied to code-reasoning tasks, including bug detection (Allamanis et al., 2018), type inference (Hellendoorn et al., 2018), code summarization (Alon et al., 2019b), program repair (Dinella et al., 2020), and code generation (Alon et al., 2020), among many others. The successful application of these techniques to a given task depends heavily on the *program representation* that encompasses relevant program features and model architecture.

There are many crucial choices involved in designing a suitable representation for a new task. Consider the example in Figure 1 which depicts an instance of an exception-prediction task (Kanade et al., 2020), whose goal is to predict the exception type in the placeholder "[??]" in the highlighted statement. Predicting the correct type AssertionError from a range of 20 pre-defined exception types in Python requires understanding the implementation of the check() function, which is defined in a different class (TestObject). Thus, the desirable context for the model goes beyond the immediate lexical neighborhood of the placeholder, possibly inside a chain of called functions.

The richness of information and extended scope inevitably imply that the relevant context may be very large. 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 larger scope to produce a small enough input to fit inside the memory of a GPU. For example, Transformers learn to reason about code from a sequence of tokens in the program; and GNNs with n layers—a hyper-parameter which, for message-passing architectures, determines how much of the graph is *reachable* from some immediately adjacent context to the task example—prune all nodes that are further than n graph hops away from the placeholder. However, distance is not always the determining factor in collecting relevant information. For instance, when trying to decide which exception type is applicable, it may be more important to follow control flow edges until they meet the raising of an exception than, say, fetching all adjacent statements without exception handling.

We outline two key design goals motivated by these considerations. First, we observe that programming languages have well-defined semantics, which ensures that relevant information (module imports such as pickle, class inheritance, inter-procedural control flow and data flow, deeper analyses like def/use chains and object escape, etc.) can be extracted via a number of deterministic analyses. So, instead of learning this kind of information indirectly from labeled data, we can make it directly

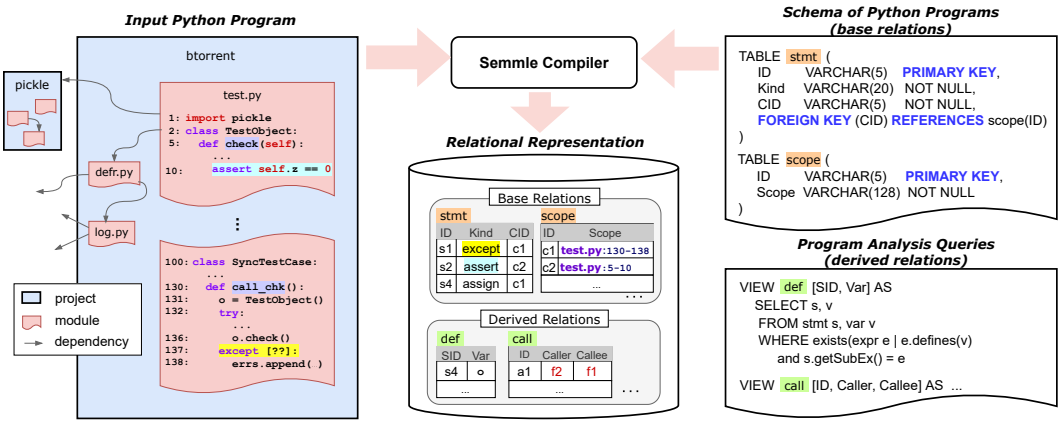

Figure 1: Example showing how CODETREK translates an exception-prediction sample in a Python program into a feature-rich representation that consists of base relations that capture the program's syntax and derived relations that capture semantic information computed by program analysis queries.

available to the model. The model can thus focus on learning the information that can only be discovered from rich contexts. Second, 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. So, instead of solely considering the model's technical constraints, we must capture relevant context in a task-specific manner.

In this paper, we propose CODETREK, a deep learning approach that realizes these goals. CODETREK leverages a declarative program analysis framework to produce a rich, easily extensible representation of context as a relational database, and a biased graph-walk mechanism for pruning that context in a task-specific way before presenting it to a model based on Transformers and DeepSets (Zaheer et al., 2017). CODETREK builds 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. CODETREK brings little modelling innovation—the architecture is reminiscent of neural techniques for knowledge-graph reasoning (Das et al., 2018; Perozzi et al., 2014). However, it is useful both as a relation generator and a task generator. As the former, it can harness existing and new analyses to add more inductive bias for hard tasks. As the latter, it can help benchmark neural techniques on more challenging tasks, and generate auxiliary training objectives to pre-train unsupervised code-understanding models (Feng et al., 2020; Guo et al., 2020; Kanade et al., 2020).

We 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. We 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.

In summary, this paper makes the following contributions:

1. We propose to represent programs as relational databases that make rich context readily available for code-reasoning tasks using deep learning.
2. We present a graph-walk mechanism that prunes the unrelated context in a task-specific manner.
3. We propose techniques to enable task designers to easily tailor and stress-test their models via program analysis queries, walk specifications, and systematic test-program generation.
4. We 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 hard tasks that follow real-world program distributions with modest effort.
5. We extensively evaluate our approach and demonstrate that deeper relational information about code helps neural models outperform the state-of-the-art in terms of accuracy and robustness.

2 THE CODETREK FRAMEWORK

2.1 BACKGROUND

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—we focus on Python in this paper, 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 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 1, we have stmt.CID $\rightarrow$ scope.ID.

2.2 A BIRD'S-EYE VIEW OF CODETREK

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. The derived information is stored in def, which, together with call, can bias the prediction of the best 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 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.

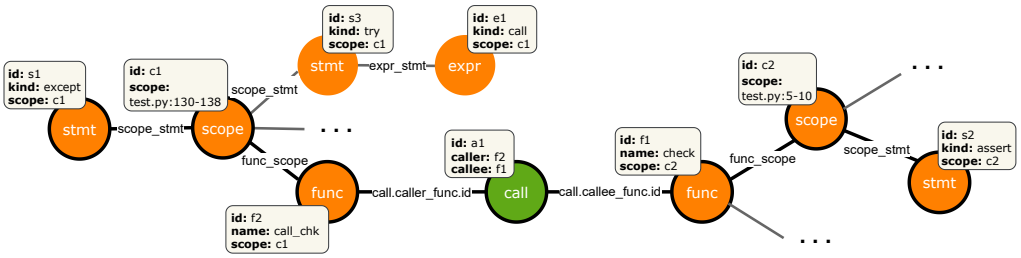

Figure 2: A partial illustration of a graph generated by CODETREK.

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 2 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 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 $\rightarrow$ 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 2, when there is a single such constraint between a pair of relations R and S, we omit the attributes from the edge type. But we do not omit them when there are multiple such constraints, such as in the case between relations func and call, namely, call.caller $\rightarrow$ func.id and call.callee $\rightarrow$ 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 the task definition. The starting node—which we 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 2, the node that represents tuple stmt(s1, except, c1)—which corresponds to the statement on line 137 in Figure 1 (left)—is the anchor because the goal of the task is to predict a suitable exception

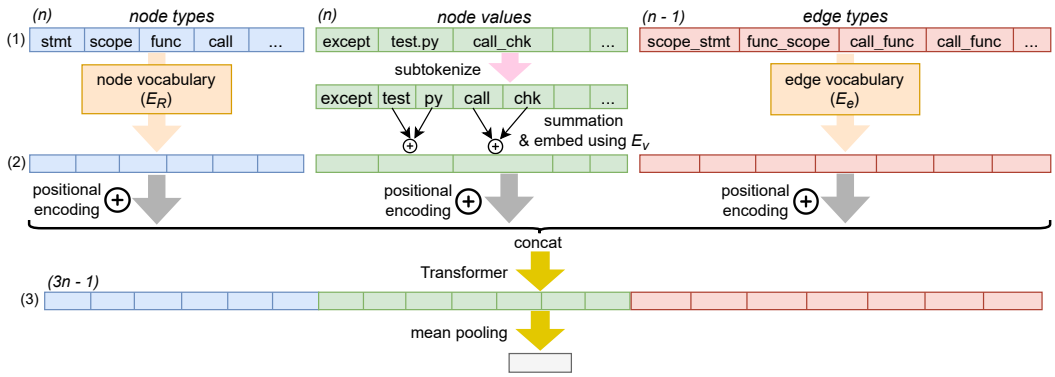

Figure 3: Embedding of the walk highlighted in Figure 2.

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 2 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 functions, the developer can assign a higher value to call nodes. In our evaluation, we assign higher probabilities to nodes of types stmt and expr. We could achieve improved accuracy for the tasks compared to baselines by only modifying the probabilities of up to 4 types of nodes. Learning the walk specification given the task without human input is exciting future work.

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, and then produces an order-invariant representation of the set of walks using the Deep Set architecture (Zaheer et al., 2017). The resulting hidden representation can then be used by 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 a transductive setting, and Code2Seq (Alon et al., 2019a) uses shortest paths between pairs of AST leaf nodes. We 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 by Hellendoorn et al. (2020). Finally, in contrast to other walk-based approaches such as Code2Seq and AnyCodeGen, which share some of our motivation, our graph structure has much richer connectivity and is larger. For instance, those approaches only consider paths ending at token-bearing leaf nodes, with only AST interior nodes in between, whereas CODETREK admits arbitrary paths through the graph. In our example, paths with more than two token nodes, e.g., the walk illustrated in Figure 2 using circles with thicker borders is admissible for CODETREK but not for Code2Seq. CODETREK builds upon the above techniques and extends to relational graphs, with different sampling strategies and neural architectures that would suit database representation better.

2.3 BUILDING BLOCKS OF CODETREK

We now describe briefly the building blocks of CODETREK. Pre-processing turns code and tasks into graphs. Then a graph is processed into sets of graph walks to present to a model. Finally, we train using a cross-entropy loss to implement a particular task.

Code as a Graph. Pre-processing 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**).

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 of a number of named tuples for each relation type. For Python, we collect 95 base relations and 277 derived relations, although adding more derived relations is simply an exercise in writing a few lines of CodeQL. Optionally, this step can limit the resulting database to only those relations that are reachable (through query-to-query dependencies) from some program analyses marked as required by the developer. Appendix C.1 provides further details.

Rel2Graph interprets the relations produced by Code2Rel as a graph, as follows. Each named tuple is represented by a node with the values of the tuple attributes as its features. Edges are added between these nodes as described in Section 2.2 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.

Neural Representation. Given a code database that is converted to graph G via the above Rel2Graph, we 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, according to the task-specific walk specification (anchor node predicate, traversal bias, and target walk length). 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. 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. Figure 3 row (1) shows such a walk representation corresponding to the walk highlighted in Figure 2. Appendix C.3 specifies Graph2Walks precisely.

Given a walk $w = [v_0, e_0, v_1, e_1, \ldots, v_{n-1}]$ of length n steps consisting of n nodes and $n-1$ edges, we produce an initial embedding $X_w \in \mathbb{R}^{(3n-1)\times d}$, where d is the embedding dimension. It consists of three parts. The first n rows of the embedding tensor correspond to the embedding of node type (the relation name) from the embedding-lookup over $E_R \in \mathbb{R}^{R\times d}$, where R is the number of relations. The next n rows are the bag-of-words embedding for the attribute sub-tokens obtained from token embedding $E_v \in \mathbb{R}^{v\times d}$, where v is our attribute-vocabulary size—we produce a WordPiece vocabulary for our attribute values. The last $n-1$ rows are the embedding lookup for the types of each edge in $E_e \in \mathbb{R}^{R^2\times d}$—recall that an edge type is the pair of relation types it connects. All three embedding matrices E_R, E_v, E_e are learnable parameters. Each individual part of the embedding tensor gets its own learnable positional encoding (denoted as $\text{PE}_w \in \mathbb{R}^{(3n-1)\times d}$). We use a Transformer encoder to represent the embedding of walk w as $\mathbf{e}_w = \text{POOLING}\Big(\text{Transformer}(X_w + \text{PE}_w)\Big) : \mathbb{R}^{(3n-1)\times d} \mapsto \mathbb{R}^d$, where after the last layer of Transformer we do mean-pooling over all $3n-1$ elements of the walk, to obtain a d-dimensional $\mathbf{e}_w$. The steps for embedding of a walk sampled from the graph in Figure 2 is illustrated in Figure 3.

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 $\{\mathbf{e}_w\}_{w\in W}$, we build a classifier by using the construction $y = \text{MLP}\big(\text{DeepSet}(\{\mathbf{e}_w\}_{w\in W})\big)$, where y denotes the predicted label and we optimize for cross entropy loss. However, for a binary classification task, we can obtain a more interpretable model via $y = \sum_{w\in W} \alpha_w \sigma(\text{MLP}(\mathbf{e}_w))$, where $\sigma(\cdot)$ is the sigmoid function, and $\alpha_w = \frac{\exp \text{MLP}(\mathbf{e}_w)}{\sum_{w'\in W} \exp \text{MLP}(\mathbf{e}_{w'})}$. This way, we can inspect the individual walks that contributed the most (i.e., the highest α_w) to the positive or negative predictions, and see if that aligns with human reasoning. We refer to α_w as the *walk score*. We train using the Adam optimizer with 8 GPUs for distributed synchronized SGD training (see Appendix A for details).

3 EVALUATION

Tasks. We 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 *declarativity*—whether the task can be stated as a logic problem that can be solved using declarative queries. For declarative tasks, we write queries in CodeQL (detailed in Appendix B); for non-declarative tasks, we rely on available datasets. We treat the following tasks:

1. VARMISUSE. Given a function and a variable accessed in it, predict whether the variable is misused. We 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 is declarative: given a well-formed program, no logic query can deterministically decide that a variable is misused, since that decision depends on the intended semantics.

Task	CODETREK	GGNN	Code2Seq	GREAT	CuBERT
VARMISUSE	**0.91** $\pm$ 0.003	0.69 $\pm$ 0.004	–	0.82 $\pm$ 0.002	0.89 $\pm$ 0.003
VARMISUSE-FUN	0.70 $\pm$ 0.004	0.54 $\pm$ 0.004	0.52 $\pm$ 0.005	**0.89** $\pm$ 0.003	0.84 $\pm$ 0.003
EXCEPTION	**0.63** $\pm$ 0.003	0.28 $\pm$ 0.02	0.30 $\pm$ 0.01	0.44 $\pm$ 0.008	0.42 $\pm$ 0.008
EXCEPTION-FUN	0.65 $\pm$ 0.01	0.51 $\pm$ 0.02	0.51 $\pm$ 0.008	0.68 $\pm$ 0.007	**0.69** $\pm$ 0.007
DEFUSE $*$	**0.98** $\pm$ 0.002	0.76 $\pm$ 0.07	–	0.84 $\pm$ 0.05	0.76 $\pm$ 0.01
DEFUSE-FUN $*$	**0.91** $\pm$ 0.005	0.77 $\pm$ 0.07	0.66 $\pm$ 0.01	0.82 $\pm$ 0.007	0.71 $\pm$ 0.01
VARSHADOW	**0.94** $\pm$ 0.007	0.71 $\pm$ 0.01	0.70 $\pm$ 0.01	0.93 $\pm$ 0.008	0.91 $\pm$ 0.008

Table 1: 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.

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. We also consider a variation of this task, EXCEPTION-FUN (Kanade et al., 2020), that is similar to EXCEPTION but takes a single function as scope. Although EXCEPTION needs inter-procedural reasoning, neither variation is declarative, since the choice of the appropriate exception type is subjective in Python.
3. DEFUSE. Given a function and a variable definition in its scope, predict whether the definition is used. We 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. Both variations are declarative (see Appendix B.1) 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. Similar to EXCEPTION, this task requires inter-procedural analysis in order to reason over both local as well as global variables. It has a declarative query (see Appendix B.2).

Benchmark. We use the ETH Py150 Open corpus consisting of 125K Python modules[1]. It is a de-duplicated and redistributable subset of ETH Py150[2]. Specifically, for the non-declarative tasks, we use the datasets released by Kanade et al. (2020). Since these are function-level samples but the EXCEPTION task is module-level, we augment the function in each sample with the entire containing module for this task. For the declarative tasks, we use analyses written in CodeQL to annotate all functions (or modules, as applicable) in ETH Py150 Open. All datasets consist of real examples, except for VARMISUSE-FUN and VARMISUSE where variable misuses are synthetically introduced into real code. We collected a number of apparent variable misuses from GitHub commits to test our models and baselines on a realistic dataset. Dataset details are available in Appendix G.

Baselines. To compare CODETREK's performance with state-of-the-art techniques, we select four baselines: we implement GGNN by Allamanis et al. (2018) and Code2Seq by Alon et al. (2019a), 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 we use ASTs as base program structures as described by Alon et al. (2019a). We sample leaf-to-leaf paths from these ASTs. The number of paths we sample is the same as the number of walks we sample for training CODETREK models. For GGNN and GREAT, we compute the data flow, 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. We detail baseline hyperparameters in Appendix A, and fine-grained information about the size of AST-based versus CODETREK graphs in Appendix F.1.

3.1 ACCURACY OF CODETREK

We evaluate the performance of CODETREK and the baseline techniques on all the tasks described in Section 3, all presented as classification tasks. We perform 10-fold cross-validation and report the average of the metric that we use to measure the performance of each task. We use ROC-AUC as the metric for DEFUSE and DEFUSE-FUN tasks due to their unbalanced datasets, and accuracy for the remaining tasks. The results are reported in Table 1. In 5 out of 7 tasks, CODETREK outperforms GGNN, Code2Seq, GREAT, and CuBERT by 2–19% points.

[1] https://github.com/google-research-datasets/eth_py150_open

[2] https://www.sri.inf.ethz.ch/py150

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 2: Robustness results of CODETREK.

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 4. 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), 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. We 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.

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

Figure 4: Example DEFUSE-FUN task.

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. We observe that GGNN fails in the presence of such long call chains on par with findings of Alon & Yahav (2021). 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 some 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.

3.2 ROBUSTNESS OF CODETREK

We 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 2.

Real-world bugs. We 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. We define a VARMISUSE-FUN bug as the occurrence of a misused variable that is changed to another in-scope variable in the commit. We 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. We 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 programs 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 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. We inspected the paths in both approaches to understand the reason behind the difference in performance of Code2Seq and CODETREK despite their similarities. We identified two reasons: 1) the kinds of program information that can be captured from an AST are limited compared to the program graph we propose, and 2) several walks that CODETREK prioritizes for this task cannot be embedded by Code2Seq.

3.3 EFFECTIVENESS ON LONGER-RANGE TASKS

We 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, we 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.

```
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 [??]:
12       log.warning()
```

Figure 5: Example EXCEPTION task.

To illustrate the kinds of mistakes that the baselines (and also CODETREK in the absence of call graph information) make, consider the representative example in Figure 5, snipped and simplified from the zipfile package. 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 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 call graph, will eventually traverse the path that reaches this raise statement from the exception statement through the call graph edges.

3.4 SENSITIVITY TO NUMBER OF SAMPLED WALKS

We evaluate the sensitivity of CODETREK at test time to the number of walks that are sampled from program graphs. All the models for the considered tasks are trained on 100 sampled walks per program. The results are reported in Figure 6. We 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.

3.5 IMPACT OF PROGRAM REPRESENTATION

To evaluate the impact of different code representations, we train two models for each task using CODETREK's architecture: for one set of models, the walks are sampled from relational graphs whereas for the other set of models, the walks are sampled from ASTs. The performance results are reported in Table 3. Notably, the models trained on walks sampled from relational graphs are about 20–36% more accurate than models trained on walks sampled from ASTs.

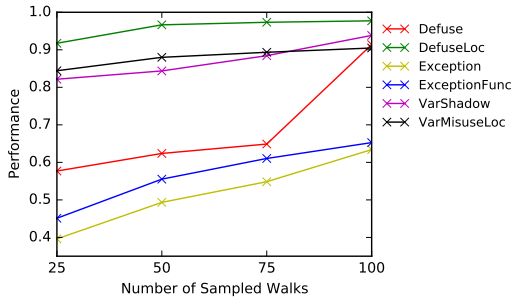

Figure 6: Sensitivity to the number of walks.

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: Impact of Program Representation on Accuracy. (Rows marked by ∗ are evaluated by ROC-AUC score.)

We also evaluate the usefulness of the ability to bias random walks in CODETREK. The accuracy results that are reported in Table 1 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, we 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. (See Appendix D for more details about specification of the tasks.) This forces the walks to only go to other functions by taking call graph edges between them. We select one of the tasks, EXCEPTION-FUN, to measure the accuracy in the absence of said biases. We re-train EXCEPTION-FUN using uniformly sampled walks and observe that the accuracy reduces from 65% to 58% as a result.

4 RELATED WORK

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 & 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., 2020; Kanade et al., 2020; Wang et al., 2021; Peng et al., 2021; Liu et al., 2020). Others have proposed to represent programs with ASTs and additional semantic edges (Allamanis et al., 2018; Brockschmidt et al., 2018) or learned abstract relations (Johnson et al., 2020), using GNN or leaf-to-leaf sequence embeddings (Alon et al., 2019a;b). Our work enables adding much richer semantic information while reducing the dependency on syntax structures. Additionally, CODETREK can take advantage of program analysis queries on relational databases to eliminate the engineering burden of augmenting program graphs with additional semantic edges.

Graph representation learning. Our 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 (GNNs) (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 & Kuang, 2021). Techniques from transductive graph embedding based on skip-gram (Perozzi et al., 2014; Grover & Leskovec, 2016) or general knowledge graph embedding (Das et al., 2017; Hamilton et al., 2018; Zheng et al., 2020) are scalable but not directly applicable for inductive setting. CODETREK achieves a good balance between modeling for large codebases and efficiency.

Feature selection techniques. 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 whereas CODETREK operates on multiple tables.

5 CONCLUSIONS

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

REFERENCES

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. A transformer-based approach for source code summarization. *arXiv preprint arXiv:2005.00653*, 2020.

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. In *International Conference on Learning Representations (ICLR)*. OpenReview.net, 2018.

Uri Alon and Eran Yahav. On the bottleneck of graph neural networks and its practical implications. In *International Conference on Learning Representations (ICLR)*. OpenReview.net, 2021.

Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. In *International Conference on Learning Representations (ICLR)*. OpenReview.net, 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*, pp. 245–256. PMLR, 2020.

Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. QL: object-oriented queries on relational data. In Shriram Krishnamurthi and Benjamin S. Lerner (eds.), *European Conference on Object-Oriented Programming (ECOOP)*, volume 56 of *LIPIcs*, pp. 2:1–2:25. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

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*, pp. 780–791. PMLR, 2021.

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

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

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.

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. *arXiv preprint arXiv:2109.08780*, 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. *arXiv preprint arXiv:1711.05851*, 2017.

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.

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)*. OpenReview.net, 2020.

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

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.

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*, pp. 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*, 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*, pp. 1025–1035, 2017.

William L Hamilton, Payal Bajaj, Marinka Zitnik, Dan Jurafsky, and Jure Leskovec. Embedding logical queries on knowledge graphs. *arXiv preprint arXiv:1806.01445*, 2018.

Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. Deep learning type inference. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 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 (ICLR)*. OpenReview.net, 2020.

Daniel D Johnson, Hugo Larochelle, and Daniel Tarlow. Learning graph structure with a finite-state automaton layer. *arXiv preprint arXiv:2007.04929*, 2020.

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and evaluating contextual embedding of source code. In *Proceedings of the International Conference on Machine Learning (ICML)*, volume 119 of *Proceedings of Machine Learning Research*, pp. 5110–5121. PMLR, 2020.

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)*, pp. 150–162. IEEE, 2021.

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*, pp. 473–485, 2020.

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*, pp. 701–710, 2014.

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.

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

Hao Yang and Li Kuang. Ccmc: Code completion with a memory mechanism and a copy mechanism. In *Evaluation and Assessment in Software Engineering*, pp. 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.

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, Ruslan Salakhutdinov, and Alexander Smola. Deep sets. *arXiv preprint arXiv:1703.06114*, 2017.

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

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*, pp. 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.

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.

A Training Parameters and Hyperparameters

In this section, we describe details on parameters and hyperparameters we used.

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

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

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

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

GGNN We train GGNN models with a learning rate of 10^{-4}, 5 layers, a latent dimension of size 128, and a message dimension of size 128.

B CodeQL Queries Used for Labeling

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

B.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(annotation_without_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 annotation_without_assignment(LocalVariable v) {
  result.getTarget() = v.getAStore()
  and not exists(result.getValue())
}

from Name unused, LocalVariable v
where
  unused_local(unused, v) and
  forall(Name el | el = unused.getParentNode().(Tuple).getAnElt() | unused_local(el, _))
select unused, v.getId()
```

B.2 VARSHADOW QUERY

```
import python
import semmle.python.types.Builtins

predicate optimizing_parameter(Parameter p) {
  exists(string name, Name glob | p.getDefault() = glob
                                | glob.getId() = name
    and p.asName().getId() = name
  )
}

predicate shadows(Name d, GlobalVariable g, Function scope, int line) {
  g.getScope() = scope.getScope()
  and d.getScope() = scope
  and exists(LocalVariable l |
    d.defines(l) and
    l.getId() = g.getId()
  )
  and not exists(Import il, Import ig, Name gd | il.contains(d)
                 and gd.defines(g)
                 and ig.contains(gd))
  and not exists(Assign a | a.getATarget() = d
  and a.getValue() = g.getAnAccess())
  and not exists(Builtin::builtin(g.getId()))
  and d.getLocation().getStartLine() = line
  and exists(Name defn | defn.defines(g)
                       | not exists(If i | i.isNameEqMain()
                       | i.contains(defn)))
  and not optimizing_parameter(d)
}

AttrNode pytest_fixture_attr() {
  exists(ModuleValue pytest | result.getObject("fixture").pointsTo(pytest))
}

Value pytest_fixture() {
  exists(CallNode call |
    call.getFunction() = pytest_fixture_attr()
    or call.getFunction().(CallNode).getFunction() = pytest_fixture_attr()
  | call.pointsTo(result)
  )
}

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

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

Table 4: Universe size of the base and derived relations that are defined by Semmle.

```

predicate first_shadowing_definition(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_definition(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
```

C CODEBASES AS DATABASES

C.1 TRANSLATING CODE TO RELATIONAL DATABASE

CODETREK views program information as relations. Table 4 shows the number of relations in three common programming languages. There is a directed acyclic graph $\mathbb{F}$ that represents the dependencies between these relations. For example, in $\mathbb{F}_{Python}$ represents the dependencies between 95 base relations plus 277 derived relations.

Example C.1. A certain Python task requires base relations $\{A, B\}$ and derived relation $\{E\}$. However, in $\mathbb{F}_{Python}$, E depends on derived relation D, which in turn depends on base relations $\{B, C\}$. Therefore, Code2Rel computes all five relations: A, B, C, D, E in order.

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 $r \rightarrow r' \in \mathsf{Edges}(\mathbb{F})$ 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) $\mathsf{Edges}(F) = \{\, r \rightarrow r' \in \mathsf{Edges}(\mathbb{F}) \mid r, r' \in \mathsf{Nodes}(F) \,\}$
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 database D and add the result to D.

C.2 TRANSLATING RELATIONS TO GRAPH

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

C.3 TRANSLATING GRAPH TO A SET OF WALKS

Definition C.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 $\mathsf{Nodes}(G)$, B is a map of bias values that correspond to each relation name, and $min, max \in \mathbb{R}^+$ specify the minimum and maximum length of the walks generated by the specification.

Example C.2. An example of a walk specification is as follows.

$\mathcal{C} = \{t | t \in \mathsf{Nodes}(G) \wedge t \in \mathsf{expr} \wedge t.kind = \mathsf{name}\}$
$B = \{\text{stmt} : 5, \text{expr} : 5\}$
$min = 3, max = 16$

Algorithm 3 (**Graph2Walks**) Given a program graph G, a 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.

C.4 BRINGING ALL THE PIECES TOGETHER

Definition C.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 C.1, and n is the number of walks to be generated.

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. $W = \mathsf{Graph2Walks}(G, T.S, T.n)$

D TASK SPECIFICATIONS

Using the notations defined in Appendix C, we describe the specifications of each task that we used for evaluating CODETREK. In the following specifications, ellipsis (...) indicate the rest of the universe of base relations as designed in Semmle framework.

$T_{\text{VARMISUSE-FUN}} = \{$

$R_B = \{\text{stmt, expr, variable, ssa-defn, ssa-use, successor, ...}\},$
$R_Q = \{\},$
$S = \{$
$\mathcal{C} = \{t \mid t \in \text{Nodes}(G) \wedge t \in \text{expr} \wedge\ t.Kind = \text{name}\},$
$B = \{\text{stmt} : 5, \text{expr} : 5, \text{variable} : 5\},$
$min = 4, max = 16$
$\},$
$n = 500$

$\}$

$T_{\text{EXCEPTION-FUN}} = \{$

$R_B = \{\text{stmt, expr, variable, ssa-defn, ssa-use, successor, ...}\},$
$R_Q = \{\},$
$S = \{$
$\mathcal{C} = \{t \mid t \in \text{Nodes}(G) \wedge t \in \text{stmt} \wedge\ t.Kind = \text{except} \wedge t.Type = \text{HoleException}\},$
$B = \{\text{stmt} : 5, \text{expr} : 5, \text{variable} : 5\},$
$min = 4, max = 16$
$\},$
$n = 100$

$\}$

$T_{\text{EXCEPTION}} = \{$

$R_B = \{\text{stmt, expr, variable, ssa-defn, ssa-use, successor, ...}\},$
$R_Q = \{\text{call-graph}\},$
$S = \{$
$\mathcal{C} = \{t \mid t \in \text{Nodes}(G) \wedge t \in \text{stmt} \wedge\ t.Kind = \text{except} \wedge t.Type = \text{HoleException}\},$
$B = \{\text{stmt} : 5, \text{expr} : 5, \text{module} : 0\},$
$min = 10, max = 24$
$\},$
$n = 100$

$\}$

$T_{\text{DEFUSE-FUN}} = \{$

$R_B = \{\text{stmt, expr, variable, ssa-defn, ssa-use, successor, ...}\},$
$R_Q = \{\text{variable-defs, local-variables}\},$
$S = \{$
$\mathcal{C} = \{t \mid t \in \text{Nodes}(G) \wedge t \in \text{expr} \wedge\ t.Kind = \text{name} \wedge t.Context \in \{\text{write, param}\}\},$
$B = \{\text{stmt} : 5, \text{expr} : 5, \text{variable} : 5\},$
$min = 4, max = 16$
$\},$
$n = 100$

$\}$

T_{DEFUSE} and $T_{\text{VARMISUSE}}$ are defined similar to $T_{\text{DEFUSE-FUN}}$ and $T_{\text{VARMISUSE-FUN}}$, respectively. However, in both T_{DEFUSE} and $T_{\text{VARMISUSE}}$ their $\mathcal{C}$ has an additional restriction $t.location =$ user selection. Also, $T_{\text{VARMISUSE}}$ is evaluated with $n = 100$.

$T_{\text{VarShadow}} = \{$

$R_B = \{\text{stmt, expr, variable, ssa-defn, ssa-use, successor, ...}\},$
$R_Q = \{\text{call-graph}\},$
$S = \{$

$\mathcal{C} = \{t \mid t \in \text{Nodes}(G) \wedge t \in \text{var} \wedge\ t.Kind = \text{name} \wedge t.is_global = \text{True}\},$
$B = \{\text{stmt} : 5, \text{expr} : 5, \text{variable} : 5\},$
$min = 4, max = 16$

$\},$
$n = 100$

$\}$

E WALK FORMAT

We describe the format of a walk as the model views it before encoding using an example. Each walk consists of three lists for node types, node values, and edges.

```
{
    "anchor": "py_stmts(415098,6,415072,2)"
    "trajectory": {
            "node_types": ["py_stmts", "py_scopes",
                "py_Functions", "py_scopes", "py_stmts",
                "py_exprs", "py_exprs", "py_scopes",
                "py_exprs", "py_variables", "v_8"],
            "node_values": [["ExceptStmt"], "",
                ["pipe","line"], "", ["Assign"],
                ["Call"], ["Attribute"],
                ["Name"], ["metr","ic"]],
            "edges": ["(py_scopes.node,py_stmts.id)",
                "(py_Functions.id,py_scopes.scope)",
                "(py_Functions.id,py_scopes.scope)",
                "(py_scopes.node,py_stmts.id)",
                "(py_exprs.parent,py_stmts.id)",
                "(py_exprs.id,py_exprs.parent)",
                "(py_exprs.id,py_scopes.node)",
                "(py_exprs.id,py_scopes.node)",
                "(py_exprs.id,py_variables.parent)",
                "(py_variables.id,variable.id)"]
    },
}
```

The node_values can consist of the value of any attribute of a node. For instance, ExceptStmt above is the kind of the node with type py_stmts, and pipeline is the name of the function that corresponds to the function which is illustrated by node with type py_Functions. The values that are identifiers, function names, etc. are subtokenized using a subword tokenizer (the tensor2tensor package). For instance, pipeline will break into [pipe,line]. Therefore, each value is encoded using the corresponding vector in the subword dictionary. If the relation name of a tuple is variable, we assign an id to it to be able to distinguish between different variables. So, the type of a variable node is determined by an id (e.g., v_8).

F ADDITIONAL INFORMATION ABOUT GRAPHS

F.1 GRAPH SIZES

Relational	EXCEPTION-FUN	EXCEPTION	VARSHADOW	VARMISUSE-FUN	DEFUSE-FUN
average	5,278	802,231	62,863	1,482	1,829
std	31,550	504,758	93,248	14,623	27,503
min	55	585	53	130	139
max	492,970	4,422,586	1,285,178	483,499	612,343

Table 5: Number of tuples (i.e., nodes) across relations of each file in the dataset used for each task.

AST	EXCEPTION-FUN	EXCEPTION	VARSHADOW	VARMISUSE-FUN	DEFUSE-FUN
average	176	3,202	667	93	177
std	244	4,117	1432	126	268
min	20	20	10	10	10
max	13,035	36,808	36,786	30,729	5,243

Table 6: Number of AST nodes for each file in the dataset used for each task.

F.2 EDGES IN BASELINE GRAPHS

GGNN Graphs. The edges that are represented to GGNN models are borrowed from Allamanis et al. (2018). They include

1. AST edges
2. NextToken edges
3. LastRead/LastWrite/ComputedFrom/LastLexicalUse edges among variable accesses
4. GuardedBy/GuardedByNegation edges between variables used in branches and their corresponding conditional expressions
5. ReturnsTo edges from the return statement to the method declaration, and
6. FormalArgName edges between method call arguments and their corresponding formal parameters.

GREAT Graphs. For GREAT models, similar to Hellendoorn et al. (2020), we borrow the edge types from Allamanis et al. (2018) and augment them with function calls.

G DATASET SIZES

The number of samples in each dataset is shown in Table 7. The VARSHADOW dataset consists of 49% samples with positive labels. The DEFUSE-FUN dataset is more skewed, with 15% positive labels, while the DEFUSE dataset has 1.6% positive labels.

Task	# Training	# Validation	# Testing	Avg. LoC (<max)
VARMISUSE	700,683	75,468	378,401	13 (<235)
VARMISUSE-FUN	700,683	75,468	378,401	13 (<235)
EXCEPTION	18,456	2,086	10,334	528 (<7,624)
EXCEPTION-FUN	18,456	2,086	10,334	32 (<1,835)
DEFUSE	217,591	52,598	104,111	12 (<528)
DEFUSE-FUN	33,182	8,149	16,296	12 (<528)
VARSHADOW	70,183	21,794	39,845	149 (<27,228)

Table 7: The number of samples used for training, validation, and testing and the lines of code that they contain. Lines of code (LoC) is reported as the average lines of code across samples in each dataset after removing the highest 0.1% and lowest 0.1% of the data.