OppropBERT: An Extensible Graph Neural Network and BERT-style Reinforcement Learning-based Type Inference System

by

Piyush Jha

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2022

Abstract

Built-in type systems for statically-typed programming languages (e.g., Java) can only prevent rudimentary and domain-specific errors at compile time. They do not check for type errors in other domains, e.g., to prevent null pointer exceptions or enforce owner-as-modifier encapsulation discipline. Optional Properties (Opprop) or Pluggable type systems, e.g., Checker Framework, provide a framework where users can specify type rules and guarantee the particular property holds with the help of a type checker. However, manually inserting these user-defined type annotations for new and existing large projects requires a lot of human effort. Inference systems like Checker Framework Inference provide a constraint-based whole-program inference framework. However, to develop such a system, the developer must thoroughly understand the underlying framework (Checker Framework) and the accompanying helper methods, which is time-consuming. Furthermore, these frameworks make expensive calls to SAT and SMT solvers, which increases the runtime overhead during inference. The developers write test cases to ensure their framework covers all the possible type rules and works as expected. Our core idea is to leverage only these manually written test cases to create a Deep Learning model to learn the type rules implicitly using a data-driven approach.

We present a novel model, OppropBERT, which takes as an input the raw code along with its Control Flow Graphs to predict the error heatmap or the type annotation. The pre-trained BERT-style Transformer model helps encode the code tokens without specifying the programming language's grammar including the type rules. Moreover, using a custom masked loss function, the Graph Convolutional Network better captures the Control Flow Graphs. Suppose a sound type checker is already provided, and the developer wants to create an inference framework. In that case, the model, as mentioned above, can be refined further using a Proximal Policy Optimization (PPO)-based reinforcement learning (RL) technique. The RL agent enables the model to use a more extensive set of publicly available code (not written by the developer) to create training data artificially. The RL feedback loop reduces the effort of manually creating additional test cases, leveraging the feedback from the type checker to predict the annotation better.

Extensive and comprehensive experiments are performed to establish the efficacy of OppropBERT for nullness error prediction and annotation prediction tasks by comparing against state-of-the-art tools like Spotbugs, Eclipse, IntelliJ, and Checker Framework on publicly available Java projects. We also demonstrate the capability of zero and few-shot transfer learning to a new type system. Furthermore, to illustrate the model's extensibility, we evaluate the model for predicting type annotations in TypeScript and errors in Python by comparing it against the state-of-the-art models (e.g., BERT, CodeBERT, GraphCodeBERT, etc.) on standard benchmarks datasets (e.g., ManyTypes4TS).

Acknowledgements

I would like to thank Professor Werner Dietl for sharing his expertise and for his constant support, guidance, and valuable advice throughout my graduate program. I would like to thank Professor Arie Gurfinkel and Professor Mark Crowley for reviewing the thesis and providing valuable feedback. I am thankful to all my colleagues in our research group, especially Zhiping Cai, Haifeng Shi and Alex Liu for their generous help and support throughout my graduate journey.

Dedication

The thesis is dedicated to my parents for their constant support and love.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software bugs have led to trillion dollars losses to the worldwide economy [59] where the glitches put a halt to the economy, expose the threat of ransomware attacks or data breaches [78], and most importantly, can take human lives [82]. Researchers have developed increasingly effective and robust testing and verification tools to prevent bugs that lead to critical run-time errors. Programming languages such as Java have a weak built-in type system that cannot avert notorious run-time errors. Type annotations can be inserted into the programs and checked at compile time to prevent such bugs. However, manually inserting these user-defined type annotations for large projects requires a lot of human effort. Pluggable type systems such as Checker Framework [22, 64] provide a sound type inference framework, but make expensive calls to SAT and SMT solvers, which increases the run-time overhead during inference. Other static analysis tools accept unsoundness as a trade-off for increased analysis speed. However, developing such tools takes considerable time and manual effort, along with a thorough understanding of the underlying type checker framework on top of which the developer would create the type inference system. Moreover, such systems must undergo rigorous testing to ensure no bugs are present within these bug-preventing tools.

Similarly, languages such as Python and TypeScript have dynamic typing, which provides flexibility to the developer, but on the other hand, are prone to run-time exceptions. Python and TypeScript introduced type annotations in their languages, helping the developers to document and maintain the code better [42] along with providing better error-detection capabilities. However, refactoring the existing dynamically typed projects to incorporate type annotations is time-consuming, error-prone, and requires substantial manual effort [37]. Static type inference techniques [26, 42] can be used to infer the supported annotations. Unfortunately, because of the dynamic features of the language and over-approximation of the program behavior, they are often imprecise [60].

The software engineering community has started to observe a recent surge in Machine Learning (ML) techniques inspired by the success of Natural Language Processing (NLP) models [9, 21, 77]. The ML-based models have a few significant advantages over the non-ML-based approaches. For example, the models are extensible and can be easily modified to extend to different type systems. Another advantage is that these models can implicitly learn the type rules from training examples (i.e., the test cases with annotations created by the developer) without the human manually needing to define the specification. Being data-driven approaches, we only need to modify the data it is getting trained on to infer for a new type system. On the other hand, ML-based methods have a disadvantage that they need a large set of training examples to accurately predict the type annotations so that the program type checks and prevents run-time bugs. To solve this problem, Reinforcement Learning (RL) can be used to fine-tune the model trained on only a few samples of test cases created by the developer. Given an extensive set of publicly available code (unannotated or partially annotated) and a sound type checker (to provide the feedback to the RL agent), the RL-loop "artificially" creates training data to better predict the type annotations, considerably reducing human effort and time.

1.1 Problem Statement

The problem addressed in this thesis is to create an ML-based type inference system that takes as an input an unannotated or partially annotated program and outputs an annotated program that is type adhering. Note that other static analysis-based methods need a developer to explicitly define the type rules or the constraint rules on top of an existing type checker to develop a type inference framework. In contrast, our model uses merely the test cases to learn the correct type rules automatically.

Specifically, we require our inference model to have the following properties:

1. First, the model must be able to automatically learn, i.e., only using a small set of test cases and type checker, the model must be able to predict annotations with no additional labeled data from the developer. The developer does not need to define the type inference rules explicitly.

2. Second, the model must be extensible, i.e., the user should be able to extend it to other type systems with minimal human effort.

3. Third, the model must be type adhering, i.e., it should produce code that type checks with high accuracy.

4. Fourth, the model must be efficient, i.e., the inference time should be less.

1.2 Brief Overview of OppropBERT

To address these challenges, this thesis presents OppropBERT, a Graph Neural Network (GNN) and Bi-directional Encoder Representations from Transformers (BERT)-style RL-based type inference model. Unlike traditional ML and non-ML-based techniques, our model has all the above-mentioned features. Our model is built on top of two recent Machine Learning architectures, namely, Transformers [83] and Graph Neural Networks [90], to capture the syntactical and semantic relations of the input.

The model takes as an input a program and a type checker corresponding to a specific property and outputs an annotated code that type checks and prevents run-time bugs by maintaining the chosen property. BERT-style models can better capture the interaction between different parts of the code as compared to their recurrent counterparts. They can learn the property the user wants to ensure without explicitly defining the type inference rules. The structure-aware loss functions combining both the graphical structure and the raw code helps learn the refinement rules effectively. Control-flow graphs (CFGs) represent the semantic information of a given input code, unlike the Abstract Syntax Tree (AST), which only represents the syntactical structure. Graph Neural Networks work well to capture the graphical structure of CFGs effectively. All these advantages together make the model type adhering and produce the annotated code with high accuracy. Supervised fine-tuning with test cases created by the developer makes the model easy to develop and extend to other forms of type inference rules. The developer-created test cases also undergo a k-fold filtering process to provide feedback to the developer to double-check a small subset of the test cases the algorithm found to be anomalous. Moreover, using an RL loop, with the GNN and BERT-based model as an agent, helps fine-tune the model further using publicly available datasets not created by the developer. Using the feedback from the available type checker, the agent can learn to predict better annotations by learning from a large and diverse set of examples without any additional manual effort, rendering OppropBERT automatic.

The ability of the BERT-style model to capture the code interactions effectively makes it easily extensible. The model can be extended to other type rules by first replacing the supervised dataset in the initial fine-tuning step and, subsequently, the type checker for the RL loop. The RL loop leverages the reward and penalty signals from the type checker to fine-tune the annotation prediction model even further. The most appealing aspect of our model is that the developer does not need to explicitly encode the type inference rules or write a specific type inference framework for every type system. The agent learns the representations using available test cases and RL feedback. The RL agent ensures that the predicted type annotations adhere to the required type rules instead of suggesting random permutations or using a stateless one that may not guarantee that the output program type checks correctly. Moreover, the model can 'transfer learn' the currently learned type inference rules to another type system, which has a small subset of the same rules as the previous type system, but fewer test cases for those subset of rules.

1.3 Contributions

- This thesis presents OppropBERT, a novel GNN and BERT-based type inference system model with structure-aware fine-tuning that is automatic, extensible, type-adhering, and efficient. To the best of our knowledge, there has been no previous work using a Reinforcement Learning (RL)-based approach for additional fine-tuning of a supervised annotation prediction model, predicting type annotations for Java programs to prevent run-time bugs using a data-driven approach, and transfer learning to a new type system with fewer training data.
- A new task of token-level error heatmap and annotation prediction for Java was created using the publicly available test cases from a popular static analysis tool.
- Extensive and comprehensive experiments are performed to establish the efficacy of OppropBERT for nullness error prediction and annotation prediction tasks by comparing against state-of-the-art tools such as Spotbugs, Eclipse, IntelliJ, and Checker Framework on publicly available Java projects.
- To illustrate the model's extensibility, we train and evaluate the model for predicting type annotations in TypeScript and errors in Python and compare it against the state-of-the-art models (e.g., BERT, CodeBERT, GraphCodeBERT, etc.) on standard benchmarks datasets (e.g., ManyTypes4TS).
- Through the thorough experiments, we provide a detailed comparative analysis with respect to different state-of-the-art techniques, perform ablation studies of the proposed model, present false positive and false negative cases found in other tools, and demonstrate zero and few-shot transfer learning capabilities of our model.

The rest of the thesis is structured as follows: Chapter 2 introduces the background, including the pluggable type system, Deep Learning, Graph Neural Networks, and Reinforcement Learning. Chapter 3 presents an overview of the architecture details of OppropBERT in all the different settings. Chapter 4 includes details regarding the comparative tools and models, benchmarks, metrics, and other implementation details. Chapter 5 presents the experiment results. Chapter 6 reviews the related work, and lastly, we conclude our work in Chapter 7.

Chapter 2

Background

2.1 Pluggable Type System

The built-in type systems in statically-typed programming languages prevent basic errors at compile time. For example, the Java compiler can find errors related to incompatible assignments, the use of variables without initialization, and unreachable code, to name a few. These built-in type systems do not check for type errors in other domains, such as preventing null pointer exceptions or enforcing owner-as-modifier encapsulation discipline.

A pluggable type system [10] helps enhance the language's type system in order to provide additional static guarantees by enforcing stronger semantics. Such type systems ingrain additional semantic understanding by providing a set of type qualifiers. Type rules can help enforce custom semantics over the qualified types. In this way, developers benefit by annotating the code with type qualifiers and using a pluggable type system to find potential bugs that the standard language compiler can not prevent. Type Checkers guarantee the particular property holds, thereby proving the absence of the selected domain-specific error.

2.1.1 Checker Framework

Checker Framework [22, 64] strengthens Java's type system by letting software developers detect and prevent errors in their Java programs. Checker Framework includes different checkers, each specific to a type of error. These checkers are compile-time tools that alert the developer about a selected domain-specific error and guarantee that such errors do not occur.

Some of the checkers included in the Checker Framework are as follows:

- Nullness Checker for preventing null pointer exceptions.
- Map-Key Checker to track which values are keys in a map data structure.
- Regex Checker to avoid the use of syntactically invalid regular expressions.
- Lock Checker to prevent concurrency and lock errors.
- Units Checker to guarantee that the arithmetic operations are performed using correct units of measurement.

The type checker that comes with Java can identify and stop numerous problems. There are some types of flaws, though, that it cannot detect. You can define the new type systems and use them as a plug-in to the Java compiler with Checker Framework. For additional details, readers can refer to the Checker Framework manual[1].

Flow-sensitive type refinement

The checker can deduce the type of the expression to be more specific (i.e., its subtype) than the declared or default one. Due to this flow-sensitive type refinement, developers do not need to specify type qualifiers on local variables inside a method body.

For example, if we declare a variable `@Nullable String var;`, dereferencing it in the next line would result in a warning from the Checker Framework highlighting a possible dereferencing of null. However, if the dereferencing happens inside an if-block `if (var != null) {...}`, it will be treated as a `@NonNull String` within the body of the if block and hence, would not result in any null deference warning.

2.1.2 Checker Framework Inference

It is often time-consuming to write all the type annotations in your code. Certain tools can be developed to automatically infer the annotations and accordingly insert them into

[1] https://checkerframework.org/manual/

the user's source code. For fields, method parameters, and method return types that lack a developer-written type qualifier for the given type system, whole-program inference infers the type qualifiers. In order to be consistent with all of its usage in the specified code, the inferred type qualifier will be the most specific. In the case of a field, this would mean that the inferred type would be the least upper bound of the kinds of expressions assigned to this field.

The Checker Framework Inference tool takes as an input a partially annotated source code along with library annotations to output a fully annotated source code [91]. The source code with library annotations is fed into the parser to generate a partially annotated Abstract Syntax Tree (AST). Constraint variables are created, and the AST is passed to a constraint generation and solver. The solver can be a Boolean Satisfiability (SAT) or Satisfiability Modulo Theory (SMT) solver. If the solver generates a satisfiable solution, the inferred annotations are inserted into the input source code to create a fully annotated source code.

2.1.3 Nullness Checker

The Nullness Checker in the Checker Framework guarantees that if no warnings are shown for a given program, then no null pointer exception will occur during runtime, in other words, the checker prevents all null pointer exceptions. The checker issues a warning if a non-`@NonNull` type is dereferenced, as it might lead to a null pointer exception. It also alerts if a `@NonNull` expression becomes null, as it could result in a misuse of the particular type. One of the misuses could be that the resulting null value can be dereferenced later on, but the checker does not complain about it. Similarly, the checker also gives a warning when a `@NonNull` field is not initialized in the constructor.

The nullness hierarchy consists of the following two qualifiers:

1. `@Nullable` annotation indicates that the type contains a null value, e.g., in the case of a `@Nullable` boolean variable, it could be True, False, or null.

2. `@NonNull` annotation indicates that the type does not include a null value, e.g., in the case of a `@NonNull` boolean variable, it could either be True or False, but never null. Therefore, dereferencing a `@NonNull` type would never result in a null pointer exception.

The Nullness Checker also supports declaration annotations. Unlike type annotations, such annotations are applied to the method instead of a type and are used to specify method behavior.

1. `@RequiresNonNull` indicates a method pre-condition. When `@RequiresNonNull` annotation is present on a method, the method expects the specified variables to be non-null during method invocation. Such annotations are useful for certain fields, which are generally annotated as `@Nullable`, but some methods require this field to be non-null.

2. `@EnsuresNonNull` indicates a method post-condition. `@EnsuresNonNull` is used to ensure that the specified expressions are non-null after the method has returned. Such annotations are useful for methods that initialize a field.

There are various error types associated with dereferencing a possibly-null reference. Examples of such types of errors used in the Checker Framework are as follows:

- `dereference.of.nullable`: Dereference of possibly-null reference.
- `iterating.over.nullable`: Iterating over possibly-null reference.
- `unboxing.of.nullable`: Unboxing a possibly-null reference.
- `throwing.nullable`: Throwing a possibly-null throwable.
- `condition.nullable`: Condition on a possibly-null value.
- `switching.nullable`: Switching on a possibly-null value.

2.1.4 Universe Type System and soft constraints

Aliasing, i.e., having multiple references of the same object, makes it more challenging to accurately create complicated object structures and guarantee certain invariants about their behavior. One such example is to allow mutation of an object through one reference that the other references can observe. It is a widespread problem in a lot of areas of software engineering. In order to control aliasing, the heap hierarchy can be structured using object ownership. Such an ownership guarantee can also enforce encapsulation, e.g., ensuring that its owner only initiates all the modifications of an object. Universe Type System [23] helps solve this problem. The static inference creates a constraint system that is provided to the SAT solver to find the appropriate annotations. The inference system is tunable, i.e., the users can indicate a preference for specific annotations or ownership structures by providing the weights to the solvers expressing the preference for certain heuristics or by providing partial annotations for the input source code. One such example of a heuristic could be to prefer deep ownership for fields. The ownership topology can be expressed by writing the appropriate ownership modifiers on the reference types. The ownership modifiers can express various ownership relations by adding them to the type parameters or type use. Examples explaining the ownership type system can be found in the Universe repository[2]. The ownership modifiers are as follows:

- `@Peer`: The referenced object is in the same context as the current object `this`.
- `@Rep`: The referenced object is owned by the current object.
- `@Any`: There exists no static information about the relationship of the two objects.
- `@Lost`: The two objects have a relationship that cannot be expressible as `@Peer` or `@Rep`.
- `@Self`: The current receiver object `this`.

It is interesting to note that every unannotated code can be said to be a legally-typed program if `@Peer` is used as default. Such a typing would denote a flat ownership structure. However, a flat ownership structure imposes no guarantees about the program's operation. Developers are usually interested in a deeper ownership structure as it gives better encapsulation and limits sharing. Another design consideration could be to influence what clients may call a particular method and add annotations to the method parameters accordingly. Therefore, the user needs to specify weights to some constraints so that the type inference system can handle such design considerations. Apart from the mandatory constraints in the type system, these breakable weighted equality constraints are added to encode a preference for a particular solution.

Let's look at how the type inference system works in this case. As described for Checker Framework Inference in Section 2.1.2, the type inference system creates the constraint variables, generates the constraints over these variables, and solves them to infer type annotations. In the presence of breakable constraints, they are translated to a weighted SAT formula. A weighted MaxSAT solver can be used in such a case to find out a solution such that the hard type system constraints are satisfied and, at the same time, the breakable constraints result in the maximum possible weight. The inferred typing for the program is translated to a concrete ownership modifier for the generated constraint variables in the program.

[2]https://github.com/opprop/universe

However, note that such a tunable type inference problem that allows breakable constraints is NP-hard [41]. Moreover, a developer must have to play around with these weight values and manually analyze the solutions to make sure of the strictness they wanted their heuristic to be enforced. The process gets complicated if the developer is interested in enforcing multiple such heuristics at once.

2.1.5 Checker Framework vs Bug-finding tools

A pluggable type-checker is a verification tool that prevents or detects all errors of a given variety. If it issues no warnings, your code has no errors of a given variety. An alternate approach is to use a bug detector such as Error Prone[3], FindBugs [32,33], SpotBugs[4], Jlint [7], PMD [19], or the tools built into Eclipse and IntelliJ. The NullAway[5] and Eradicate[6] tools are more like sound type-checking than bug detection, but all of those tools accept unsoundness — that is, false negatives or missed warnings — in exchange for analysis speed. For example, here are the two popular bug-finding tools:

Eclipse

Eclipse comes with a null analysis[7] to detect potential null pointer related errors in your code. However, Checker Framework's Nullness Checker is more precise, i.e., it gives fewer false positives than Eclipse. Eclipse on the other hand with a tight IDE integration is faster, mostly because it has fewer features than Checker Framework. This makes Eclipse more useful for bug-finding than for verification. Moreover, Eclipse also does not support a lot of features present in Checker Framework, such as pre- and post-conditions, polymorphism, and dataflow analysis, to name a few. Eclipse is also not extensible, while Checker Framework supports over 20 type checkers.

NullAway

NullAway is a type-checker for null pointer errors. It is lightweight (hence, faster), but unsound, i.e., no warnings does not mean that your code would not crash with a null

[3]https://errorprone.info/
[4]https://github.com/spotbugs/spotbugs
[5]https://github.com/uber/NullAway
[6]https://fbinfer.com/docs/next/checker-eradicate/
[7]http://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2Ftasks%2Ftask-using_null_annotations.htm

pointer exception. NullAway makes unchecked assumptions regarding getter methods and also assumes that all objects are always fully initialized.

2.2 Deep Learning

The machine learning tasks can be categorized as supervised, unsupervised, or semi-supervised learning [27]. In the case of supervised learning, a given set of input-output pairs is used to train the model so that it can make predictions of an unseen dataset. Training a model means that given an input feature vector, the model learns the underlying function to map it to the corresponding target labels.

Classification and regression are two common tasks in the domain of supervised learning. For classification problems, the model's objective is to predict the category C to which the input belongs. The model learns a function $f : \mathbb{R}^n \to \{1, ..., C\}$, where n is the dimension of each input data [27]. For example, the Twitter sentiment analysis problem is a classification task, where the model is given a dataset of tweets along with the sender details and other meta information and is required to predict whether the tweet has a positive or negative sentiment.

Regression tasks, on the other hand, output numeric values. The model learns the function $f : \mathbb{R}^n \to \mathbb{R}^m$, where n and m are the dimension of the input and output data, respectively [27]. For example, stock price prediction models are based on regression. The model takes as an input the history of the particular stock along with other meta-data about the company to output the stock price prediction.

In the case of unsupervised learning, the aim is to figure out the hidden patterns from an unlabeled dataset. Usually, they either try to learn the probability distribution of the dataset generator or find a way to segregate the dataset into different clusters. Recommendation engines use clustering algorithms to suggest similar results (e.g., song recommendation in the case of Spotify) to users with similar interests.

Another closely related approach to unsupervised learning is self-supervised learning, which can be considered as an intermediate form between supervised and unsupervised learning. Self-supervised learning is similar to unsupervised learning, except that the end goal of a model trained using self-supervised learning is ultimately to help a supervised learning algorithm.

In order to evaluate the performance of a machine learning algorithm, we would have to use a performance measure. The choices usually depend on the task at hand. Since our problem deals primarily with classification, we will focus more on that in the following section. Most of the definitions in this section are based on the Deep Learning book [27], and we refer the readers to this wonderful book for further reading.

2.2.1 Classification task

One of the most common metrics to measure the performance of classification models is accuracy. Accuracy is defined as the percentage of correctly predicted labels over the total number of predictions. Similarly, an error rate is the ratio of incorrectly predicted labels over the total number of predictions. Given below, we define other popular metrics, namely, precision, recall, and specificity:

The precision of a model is defined as follows:

$$Precision = \frac{TP}{TP + FP}$$

The recall of a model is defined as follows:

$$Recall = \frac{TP}{TP + FN}$$

The specificity of a model is defined as follows:

$$Specificity = \frac{TN}{TN + FP}$$

Let us now define the abbreviations used in the above formulas. Consider the case of an e-mail spam filter. True positive (TP) cases are where the filter predicts as spam (yes) and the e-mail is actually spam. Similarly, true negative (TN) implies that the filter predicts as not spam (no) and the e-mail is indeed not spam. A false positive (FP) is the case where the filter predicts the e-mail to be spam, but the e-mail is not spam. Similarly, false negative (FN) denotes that the filter does not predict the e-mail to be a spam, however, the e-mail was indeed spam.

Depending on the use case, users tend to trade off on precision, recall, and specificity. For example, if we want to avoid false positives, we would prefer high precision. However, if we want higher coverage and do not want to miss a positive case, we would prefer a higher recall, let's say to test for a highly contagious virus such as COVID-19, as we aim to minimize the infection rate. Similarly, if we want to avoid false alarms, we would prefer optimizing specificity, e.g., cancer diagnosis, as it costs a lot of money to the patient and there are side-effects to the treatment.

Note that the performance of machine learning algorithms should be evaluated on data it has not seen before. In order to make this possible, standard practice is to split the dataset into training and test sets, where the training set is usually the more prominent in size. As the name suggests, the training set is used by the algorithm to train/learn, while the test set is used to evaluate the performance of the machine learning model.

Training the model involves creating a loss function and then using an optimization algorithm to find the best solution. In supervised learning, the objective is to minimize the loss function, which measures the difference between the predicted and the ground truth values. A higher loss signifies poor predictions, while a lower one implies that the model is learning well. A popular choice of a loss function for classification tasks is the cross-entropy loss, which is based on maximizing the log-likelihood estimation of data for a given model.

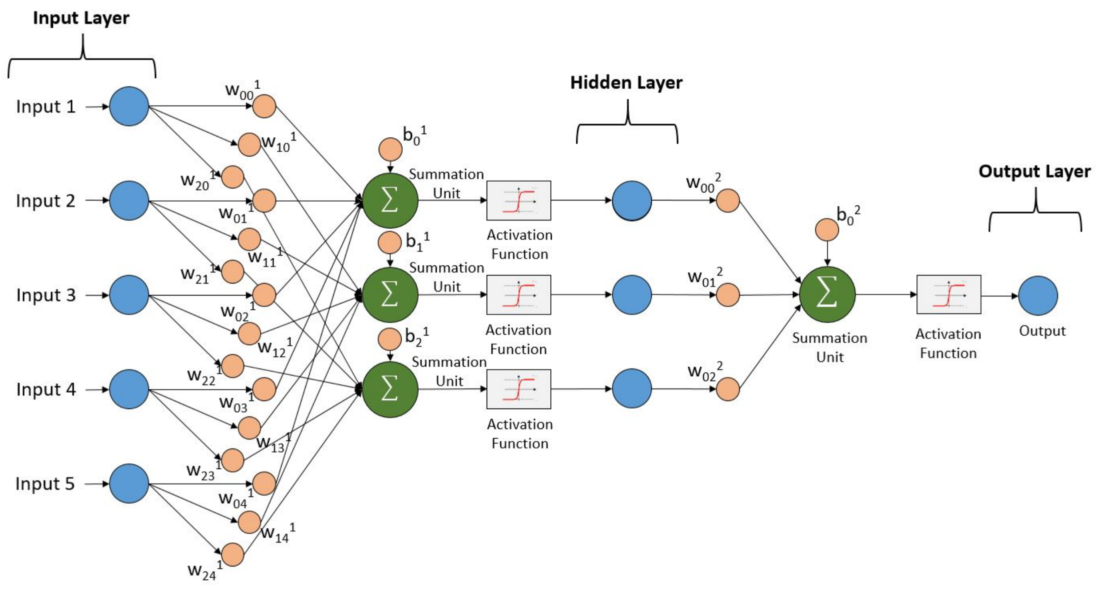


Figure 2.1: A Neural Network

2.2.2 Parameter updates

Figure 2.1 shows a typical neural network with neurons connected in the form of an acyclic graph. The network consists of distinct layers of neurons. The input layer receives the input to the model, which is then passed on to the hidden layer(s) and finally to the output layer. The neural network is initialized with random weights, which are updated as the learning progresses.

Gradient-based optimization algorithms can be used to minimize the loss of a predictive model with regard to a training dataset. For calculating the gradients of the weights in a neural network, a back-propagation algorithm can be used. Both gradient descent and back-propagation algorithms are used together for training a neural network. Merely a simple stochastic gradient descent cannot be used if we want to calculate the gradient of the nodes in the hidden layer. Back-propagation is an efficient algorithm that implements the chain rule to calculate the gradients for any parameter in the neural network. The optimization algorithm then uses this gradient to update the model weights.

2.2.3 Overfitting and Regularization

By increasing the number of learnable parameters in a neural network, the model can often better fit the training dataset. Overfitting is the phenomenon when there is a considerable disparity between the training and test error. To prevent overfitting, different regularization techniques can be used. One idea is to decrease the number of learnable parameters or restrict the parameter range [27]. The notion of regularization is derived from Occam's razor. According to the principle, we choose the most straightforward hypotheses from all the different ones that explain a particular observation equally well. From a Machine Learning perspective, this means that more preference should be given to simple or, in other words, sparse parameter sets and thus, penalizing highly complex models. L1 regularization is one way to achieve this, as it favors sparse matrices. That's why it can also be viewed to perform feature selection. L2 regularization or weight decay is another popular choice that makes use of the Frobenius norm on the weight matrices. Dropout can also be used while training the neural network to prevent overfitting. In this method, the neurons are randomly and temporarily disabled with some user-defined probability p.

2.2.4 Transformers and BERT Models

The input to our problem is in the form of a text. They are sequential and also have variable lengths. Recurrent Neural Networks (RNNs) such as Long Short-Term Memory

(LSTM) with attention has been a popular choice for a long time to learn the long-range temporal dependencies for varying length inputs. However, a recently evolved technique called Transformers [83] uses an architecture relying entirely on self-attention without using any sequence-aligned models.

We use a Bi-directional Encoder Representations from Transformers (BERT) model [83] as an actor-critic agent in an RL setting. Transformer-based models are critical components in the hugely successful Natural Language Processing models such as GPT-2 [68], GPT-3 [12], and Google's PALM [18] etc. More recently, they have been successfully applied in the context of formal programming language applications such as code translation [43,58], code synthesis [5,14], code understanding [25,29,62].

Bi-directional Encoder Representations from Transformers (BERT) models are based on the Transformer architecture [21,53]. BERT models take as input (textual) strings and encode them into a vectorized representation. One aspect of BERT that is particularly relevant to us is that they are able to learn representations of formal grammars, as can be seen by their efficacy in above-mentioned applications such as code translation, code synthesis, code repair, etc. For more details, we refer the reader to the paper by Bommasani et al. for a comprehensive overview of BERT models [9]. For additional details and context, we suggest reading papers by Devlin et al. [21] and Liu et al. [53].

The input text is represented as a vector of embeddings. Positional embeddings are added to these embeddings to capture the positional aspects of the input with respect to each other. A self-attention mechanism is used to compute the representation of the given sequence by looking at different input sequence positions. In terms of RNNs, they can be seen as the hidden states which represent the previous embeddings along with the current one. The self-attention model integrates the interactions of the relevant words within the present word. The attention function takes as input a query with a set of key-value pairs to produce an output vector. The output is a weighted sum (as a function of the query and the corresponding key) of the values. Additionally, a multi-head module runs the attention mechanism multiple times in parallel. Using a multi-head structure allows the model to project the input into different representation subspaces to encode additional contextual information and relationships for each word.

2.3 Graph Neural Network (GNN) training

Graphs are ubiquitous and the definitions of actual world objects frequently depend on how they link to other entities. A graph is a natural way to represent a collection of entities and the relationships among them. Researchers have created neural networks utlilizing graph data that are called Graph neural networks, or GNNs).

Let $G = (V, E)$ denote a graph where $V = \{v_1, v_2, ..., v_n\}$ and $n = |V|$ is the number of nodes in the graph. Let the feature vector of node v_i be x_i. GNNs are designed to learn how to represent nodes and graphs. Modern GNNs often use a learning schema that aggregates the representations of a node's first or higher-order neighbours to iteratively alter the node representation.

2.3.1 Mini-batching

To train a deep learning model to scale to large amounts of data, mini-batch creation is essential. A mini-batch aggregates a number of samples into a single representation so they can be processed efficiently in parallel as opposed to being processed one at a time. This process is often accomplished in the area of images or languages by padding or rescaling individual examples into sets of equal-sized shapes, after which the examples are grouped in another dimension. The length of this dimension is therefore known as the batch size and is equal to the number of samples combined in a mini-batch.

We can employ mini-batching to manage a GNN's memory footprint. In each batch, it is intended to run the node-level message passing equations for a portion of the graph's nodes. Careful engineering can be used to make sure that we only compute the embedding for each node in the batch a maximum of once. This method does have a huge drawback, though. That is, message forwarding cannot be implemented directly on a subgraph without information being lost. Every time a node is eliminated, its edges are also eliminated. Therefore, even though two nodes are connected in the whole graph, it is feasible that they are no longer connected in the subgraph. To get around this restriction, a number of methods have been suggested, such as subsampling node neighborhoods. It is expensive to just use the entire graph for training, especially for dense graphs. Additionally, because each graph is unique, adding previously unknown nodes to a graph would need restarting training.

The standard mini-batching approach is either impractical or may lead to excessive memory consumption because graphs are one of the most general data structures and can contain any number of nodes or edges. Using Pytorch Geometric, we choose a different strategy to do parallelization over numerous cases. Here, adjacency matrices are stacked diagonally to form a large graph that contains numerous isolated subgraphs, and the node and target attributes are simply concatenated in the node dimension. Since messages still cannot be sent between two nodes that are a part of different graphs, GNN operators that rely on a message passing scheme do not need to be altered. Both computational and memory overhead is eliminated. This batching method, for instance, functions flawlessly without any padding of node or edge features. Adjacency matrices are saved sparsely, containing only non-zero entries, or the edges, therefore there is no additional memory overhead.

Following a convolution layer in convolutional neural networks (CNNs) is typically a pooling layer. The goal of layer pooling is to obtain more universal features. Because CNNs are very effective at handling images, there has been a lot of work done to adapt pooling modules to graph structures. The node-wise aggregation operators, such as max, mean, and sum, are still among the most common options for creating graph pooling modules since they are a natural extension of CNNs.

Augmentation is the process where a graph is enriched by adding new nodes, edges, or features. When the input graph lacks attributes, we can give each node a distinctive ID (one-hot vector). We can manually add any additional properties that may be relevant, such as node degrees, cycle counts, or clustering parameters. Contrarily, graph augmentation involves adding nodes or edges to existing graphs. The procedure of passing messages via a sparse graph could be challenging. Then, all we need to do is add more virtual edges.

2.3.2 Regularization on GNN

Graphs can use several of the common regularization techniques, such as dropout and L2 regularization. Additionally, there are regularization techniques tailored specifically for GNNs. One such technique is parameter sharing across layers. For a network with multiple layers, it is possible to share a specific set of parameters across layers. For instance, the Gated Recurrent Units (GRUs) of different layers employ the same set of weight matrices. The number of parameters increases linearly with the number of layers, making training for large graphs challenging. Moreover, multi-relational GNNs frequently employ this parameter-sharing strategy. The number of learnable parameters is further inflated by the fact that the number of parameters in these GNNs rises linearly with the number of relation types. The possibility of vanishing or exploding gradients must be carefully considered while employing this method. Utilizing edge dropouts is an additional option. In order to make the network more resistant to noise in the adjacency matrix, it is intended to randomly mask edges during training.

2.4 Reinforcement Learning

2.4.1 Deep Q-Learning

There is a large literature on reinforcement learning, and we refer the reader to the excellent book by Sutton et al. [80] for further reading. Deep Q-Learning is a Reinforcement Learning algorithm in which control policies are learned by interacting with an environment E with the help of an agent. Given a set of internal states and a predefined set of actions, the agent performs a certain action A, in the given state S to transition to a new state S'. By performing this action, the agent will receive a certain reward R from the environment E. The agent aims to maximize the cumulative reward by performing a sequence of actions. [30, 76]

These actions result in a set of transition tuples (S, A, R, S') of a Markov Decision Process (MDP). Q-learning [87] is a popular model-free RL algorithm where the agent tries to learn an optimal expected long-term reward of an action, denoted as the value function $Q(S, A)$. A model-free algorithm explores the environment and learns from the outcomes of the actions without creating an internal model of the environment. Q-learning is an off-policy algorithm and it estimates the reward for a given state-action pair using the greedy optimal policy. The optimal action-value function is approximated independently of the policy. Since Q-value relies on all the possible state-action pairs that would be unpragmatic, a parametrized version of the function can be used to approximate $Q(S, A)$. The learned parameters can generalize over the states and actions involved in the given environment [11, 79]. The advancement of Deep Learning techniques and their capability of function approximation of a wide range of complex problems led to the introduction of Deep Q-Network (DQN) [61]. The Q-values keep on updating by performing the actions suggested by the present Q-value function.

In order to improve the agent's knowledge, one needs to take care of the exploration-exploitation trade-off. If the agent has not explored sufficiently and tries to exploit early, it might give us rewards from a sub-optimal behavior. Exploration would help the agent receive better rewards in the long run. The epsilon-greedy method to tackle this trade-off. Furthermore, if the transitions from only the current input are used to train the Q-value model, the model would overfit, and the same action would be repeated for any new input. To avoid this correlation problem, we can randomly sample the transition tuples from a pool of past and current replay memories. A target network can also been used to make the learning process stable.

2.4.2 Proximal Policy Optimization (PPO) Algorithm

As mentioned in the previous sub-section, Deep Q Network (DQN) [61] has been one of the most popular RL algorithms in which control policies are learned by interacting with an environment with the help of an agent. DQN uses deep learning techniques for function approximation of the optimal expected long-term reward of an action, i.e., the Q function. For a large number of state-action pairs, it is difficult for DQN to learn the Q function. Proximal Policy Optimization (PPO) was proposed by OpenAI [73], which handles this drawback using a Clipped Surrogate Objective function. PPO provides a better convergence and performance rate as compared to other techniques [73].

Let's start by introducing the loss function for a simple policy gradient method. The policy gradient loss in a policy objective function is defined as the expectation of the log of policy times the advantage function. The policy is a model that produces an action for a given state. The advantage value is the difference between the current state's discounted rewards and the predicted final return value. A positive advantage value shows that the agent performed better than an expected average return. The gradient update would be made so that agent would select the current action if we encounter this state. Similarly, a negative advantage means that this action should not be taken in the future when you are in this state.

$$L^{PG}(\theta) = \hat{\mathbb{E}}_t[\log \pi_\theta(a_t|s_t)\hat{A}_t] \tag{2.1}$$

However, the major drawback of the policy gradient method is that it might converge very slowly if the step size is small, and there will be a drastic variability in training for a larger step size. A second drawback is that a single batch would ruin the policy and would not be able to generalize properly if the new policy is drastically different from the older one. To overcome the latter, a KL-divergence constraint was added by Trust Region Policy Optimization (TRPO) [72], but this has a high computational cost involved with the calculation of KL-divergence.

$$L^{CPI}(\theta) = \hat{\mathbb{E}}_t\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\hat{A}_t\right] = \hat{\mathbb{E}}_t[r_t(\theta)\hat{A}_t] \tag{2.2}$$

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t[\min(r_t(\theta)\hat{A}_t, \mathrm{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)] \tag{2.3}$$

Here, the probability ratio $r_t(\theta)$ denotes how likely the action is now compared to the older policy. A value greater than 1 indicates that the action is more likely now than in the older policy, and a value between 0 and 1 indicates that the action is less likely now. Multiplying it by the advantage value would make the objective function L^{CPI} similar to TRPO. However, L^{CPI} is unconstrained, and thus, it can give rise to large policy updates. Now, by including clipping in L^{CLIP}, the probability ratio is between $1-\epsilon$ and $1+\epsilon$. This ensures that the update in the policy does not change it much from the older policy, leading to less variance for a smooth training and ensuring that the agent does not wander off to an unrecoverable path because of a drastic policy change. In order to give an intuition of why it works, let's consider the case that a certain action is less rewarding with the new policy compared to the older one. Instead of bringing the likelihood down to zero with merely one such instance and ruining the policy learned so far, limiting the gradient update would be better and allow the agent to explore better. An entropy term is also usually added to ensure that the agent has performed enough exploration.

Unlike replay memory, as used by DQN, PPO collects a small batch of transition tuples to update the policy once per episode/epoch. In the next episode, the old batch is discarded, and a new batch is collected with the policy updated in the last epoch. This is why PPO is on-policy learning, unlike off-policy learning of DQN, because the accumulated experience is only used once to update the current policy.

Chapter 3

Overview

In this section, we present an overview of the architecture details for the three different settings: annotation prediction, Reinforcement Learning-based fine-tuning, and error heatmap prediction.

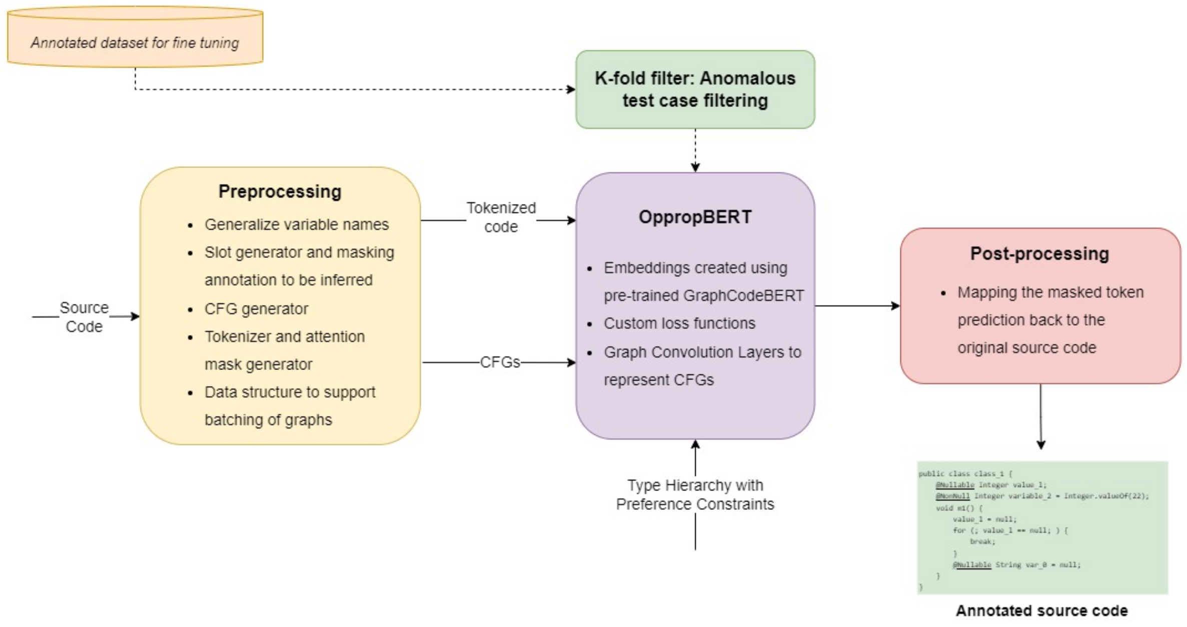
3.1 Annotation Prediction

Given an input file and type hierarchy (if one exists) with preference constraints, OppropBERT for annotation prediction produces a type-adhering annotated file. Leveraging the data-driven idea of using test cases created by the developer to fine-tune the pre-trained model in a supervised manner, OppropBERT can learn the type rules from the test cases and predict type annotations with high accuracy. The architecture diagram is shown in Figure 3.1.

Input: A program (the whole file or a block of code) written in a particular programming language to be annotated is given as the input. The file could either be unannotated or partially annotated. Additionally, the user provides the list of possible types in the type system and the type hierarchy (if one exists). The user can also define preference constraint weights for different locations, e.g., return types, parameter types, etc.

Output: An annotated file that adheres to the type rules so that it can guarantee a defined property of choice and prevent run-time bugs.

K-fold filtering for supervised fine-tuning: The model utilizes the developer-created test cases with annotated code for fine-tuning the Deep Learning (DL) model to


Annotated dataset for fine tuning
K-fold filter: Anomalous test case filtering
Preprocessing
Generalize variable names
Slot generator and masking annotation to be inferred
CFG generator
Tokenizer and attention mask generator
Data structure to support batching of graphs
Source Code
Tokenized code
CFGs
OppropBERT
Embeddings created using pre-trained GraphCodeBERT
Custom loss functions
Graph Convolution Layers to represent CFGs
Type Hierarchy with Preference Constraints
Post-processing
Mapping the masked token prediction back to the original source code
Annotated source code

```
public class class_1 {
    @Nullable Integer value_1;
    @NonNull Integer variable_2 = Integer.valueOf(22);
    void m1() {
        value_1 = null;
        for (; value_1 == null; ) {
            break;
        }
        @Nullable String var_0 = null;
    }
}
```

Figure 3.1: Architecture diagram of OppropBERT for annotation prediction task.

predict the correct annotations. However, the developer-created dataset may have some bugs due to human error. DL models are sensitive to noisy data, and thus, the developer must make sure that the test cases to be used as the fine-tuning data must be perfect so that the model learns the correct type rules. Utilizing a filtering mechanism to find such anomalous test cases so that the developer can double-check a certain fraction of them can help prevent any negative impacts on the model.

A k-fold cross-validation approach has been used to filter out the dataset. For our experiments, we have chosen k to be 5. The dataset of the test cases is divided into k non-overlapping subsets. On trial i, the i-th subset is used as the validation set, while the remaining $(k-1)$ sets are used for training the error prediction model using the OppropBERT model explained later. Post training, the validation set is used to find the probability of error for each test case; let's call the output to be y_{pred}. The process is repeated for all the folds. The developer can choose to double-check either the top n percent of the validation data (i.e., the ones with the highest probability of error) or choose from a certain threshold of y_{pred}.

Furthermore, the training set was also deduplicated to avoid any bias using the method proposed by Allamanis [2].

Preprocessing: We generalize the keywords present in the code similar to the obfuscation process used by DOBF [44] as this helps the DL model not to create any bias from the variable names in the developer-created test cases. The slots are created in the modified code to represent the locations where annotation can be inserted. These slots are replaced with the mask token so that the annotation prediction model only focuses on predicting the annotations for the masked-out locations. The modified input code is fed into a Control Flow Graph (CFG) generator. In the case of Java, we use the Checker Framework CFG builder, which transforms the Abstract Syntax Tree (AST) of each method into a CFG. Following this, a pre-trained GraphCodeBERT [29] tokenizer is used to tokenize the input code and CFG nodes. The tokenizer also takes care of padding the mini-batch of input and creates the attention mask to prevent the model from focusing on the padding tokens while calculating the self and cross-attention scores. Lastly, all the pre-processed elements of the code are stored in our multi-graph data structure. Apart from whole program inference, we randomly mask a subset (25-50%) of the existing annotations to allow the model to learn to predict the missing (i.e., masked out) annotations in the given partially annotated code.

OppropBERT: The tokenized code and CFGs are fed into the OppropBERT. The pre-trained GraphCodeBERT [29] model is used to create embeddings (feature vector representation) for the nodes of the CFGs and the raw (masked) code. The node embeddings of a graph are passed to Graph Convolution layers. Lastly, we obtain the graph embedding using the Global Mean Pooling of all the nodes present in the graph. The same procedure is repeated for all the available graphs for a particular program. The code and the graph embeddings are concatenated together and passed to a fully connected layer to obtain the required probability distribution across the possible annotations. Appropriate regularization is used at every stage of the network, chosen with the help of the performance on the validation set. During pre-training, GraphCodeBERT uses a standard masked language modeling task [21] to learn the source code representations, data flow edge prediction to learn the representation from the dataflow by predicting the masked out edges, and lastly, a variable-alignment task across source code and dataflow.

Loss functions - The masked language modeling (MLM) task works as follows. We start by randomly masking 15% of the tokens of the code and replacing them with the masked token 80% of the time, with a random token 10% of the time, and leaving them as it is the remaining 10%. Using the MLM objective, the model is trained to predict the original tokens to learn the representation space better, as shown in [21,25,53]. We modify the MLM objective to consider both the source code and the CFG nodes as depicted in

Equation 3.1. To solve the masked token classification problem, we use the Cross-Entropy Loss function, a standard loss function for classification problems. We take into account the class imbalance and masked token weights since we are interested in classifying only a subset of tokens that are marked as masked. Additional user-defined heuristics, such as preferring a subset of annotations for certain locations, preferring a particular code pattern, or an ownership structure are also incorporated in the loss function.

$$\mathcal{L}_{cfg\_MLM} = -\frac{1}{K}\sum_{k=1}^{K} \log p(x_{mask} \mid X_{code}, X_{cfg}; \theta) \tag{3.1}$$

We also introduce additional loss functions to learn the representation better:

1. Node-code association: To better learn the representation from CFG, a loss function is defined that improves alignment between the source code and the CFG node, as shown in Equation 3.2. Positive and negative samples are sampled equally to avoid any bias. A_{node_code} denotes all the association pairs between node-code. $\delta(a_{nc} \in A_{mask})$ is 1 if the particular a_{nc} association exists between the CFG node and the source code, else 0. $p_{a_{nc}}$ is the probability of the association calculated by combining the embeddings from OppropBERT.

$$\mathcal{L}_{association} = -\sum_{a_{nc} \in A_{node\_code}} [\,\delta(a_{nc} \in A_{mask}) log p_{a_{nc}} + (1 - \delta(a_{nc} \in A_{mask})) log(1 - p_{a_{nc}})] \tag{3.2}$$

2. Flow-sensitive refinement: Flow-sensitive type refinement helps the type annotations to be refined at different places in the code instead of assigning a single type annotation for a particular variable throughout the code. The refinement depends on the use of the variables at different locations of the code, which the CFG captures well. x_{π_k} denotes the mask token. Unlike the standard MLM described above, the flow-sensitive refinement loss predicts the masked token only by looking at the nodes and source code before this masked token.

$$\mathcal{L}_{flow} = -\frac{1}{K}\sum_{k=1}^{K} \log p(x_{\pi_k} \mid X_{-\pi}; \theta) \tag{3.3}$$

Tokenizers - Word embeddings often represent words in the corpus and capture the semantic and syntactic similarity of different words in a common subspace. However, there are many out-of-vocabulary (OOV) words, as we cannot expect all the words possible in a given language to be present in the training corpus. Also, the presence of rare words does not allow the model to learn proper word embeddings. Character and sub-word level representations can help address these drawbacks. Authors in [47] developed a character-to-character model, which worked better than a word-to-word translation model because of its ability to tackle OOV and rare words. However, it had limitations. The attention layer's computational cost increases quadratically with the increase in sentence length due to using character-based tokens. Even at the encoder, the character-level encoder needs to learn a complex non-linear function for a long sequence of characters and capture dependencies over a longer time span. GraphCodeBERT [29] tokenizers are based on a BERT tokenizer[1] which uses WordPiece, first introduced by Schuster et al. [74]. WordPiece initializes the vocabulary and includes every character present in the training data. It then progressively learns a given number of merge rules. Instead of choosing the most frequent symbol pair (as used in Byte Pair Encoding [75]), it chooses the one that maximizes the likelihood of the training data once added to the vocabulary. In other words, it evaluates the loss by merging two symbols to ensure it's worth it. We use the GraphCodeBERT tokenizer as it is pre-trained on CodeSearchNet dataset [38], which includes 2.3 million functions in six programming languages (Go, Java, JavaScript, PHP, Python, and Ruby). Moreover, we also add the type qualifiers as single tokens in the tokenizer vocabulary so that they do not split into multiple sub-words. Otherwise, it will create problems during MLM due to the presence of multiple masks for the same slot location. Adding to the token embeddings means extending the embedding matrix to accommodate the new tokens in the vocabulary. Instead of initializing them randomly, we take the average across the individual tokens from the existing pre-training tokenizer.

Post-processing: During the post-processing stage, the predicted annotations by OppropBERT are mapped back to the source code locations to generate a file. By playing around with the preference weights provided in the input, the final annotated file could be made less or more conservative. If no solutions exist (the combined confidence score is low for all the predictions), the model can remove existing annotations (added by the user in their partially annotated code) to predict appropriate ones. Similarly, the user can prefer a specific annotation to occur more frequently than others, e.g., `@Rep` in universe type system [23] to enforce a deeper ownership structure. This can be done by giving a higher weightage to predicting the preferred annotation while ensuring that the final program is type-adhering.

Extending to other type systems: For a given programming language (let's say Java), if the type hierarchy and rules of the current type system on which the model is

[1] https://www.tensorflow.org/text/api_docs/python/text/BertTokenizer

trained are similar or a superset of the new one with less training data, then the existing model can transfer learn the new type system easily. Replacing the old annotation tokens with the new ones in the vocabulary would preserve the embeddings associated with the relative position and hence, the type rules associated with the old ones.

Extending to other programming languages: Since the pre-trained model is programming language agnostic, the same model architecture can be easily extended to any new programming language and its type system. We have chosen Python and TypeScript to demonstrate this ability.

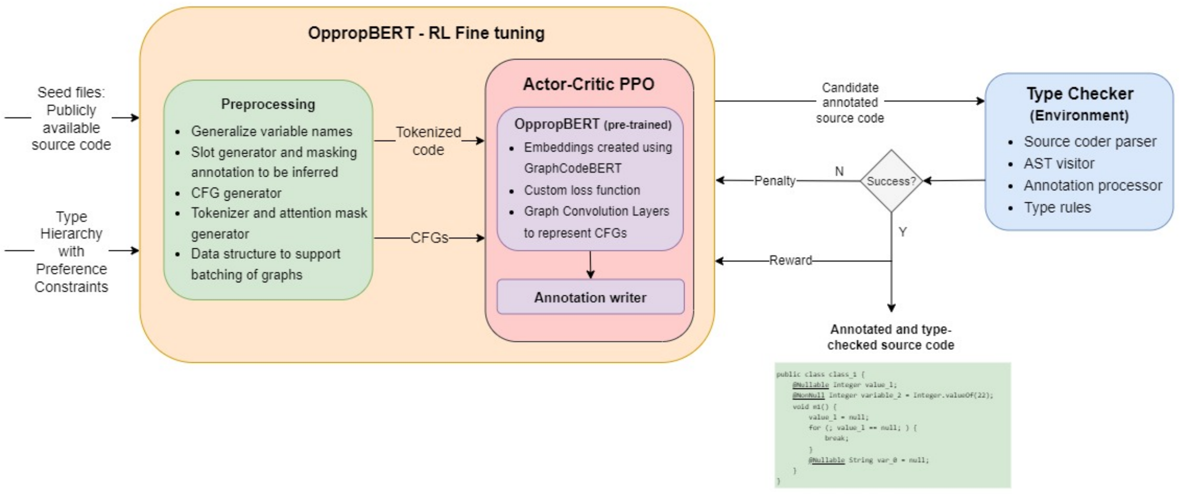
No fixed point iteration: Since we use a self-attention network, we can parallelly obtain all the predictions for different slots. This especially helps us with annotation prediction, where we do not need to iterate multiple times over the program as in the case of fixed point iteration [35, 36]. In a fixed point iteration, unannotated variables are initialized to the maximal set of qualifiers. The analysis would iterate over the program statements refining the initial sets in every pass till the fixed point is reached. In our case, we use beam search decoding to find the best possible combination of all the available choices of annotations in a single pass.

3.2 Reinforcement Learning fine-tuning

In the previous section, we leverage the idea of using test cases created by the developer to create a supervised fine-tuning model to predict type annotations. However, the limited number of test cases might not be good enough to generalize for the entire type system. Given additional unannotated code written in the programming language under consideration and a type checker corresponding to the annotations of the property we are interested in inferring, a Reinforcement Learning (RL) agent can be used to fine-tune the model defined in the above section and predict annotated and type-checked files. The architecture diagram is shown in Figure 3.2.

Input: Publicly available files from Google Big Query are used as seed file input to the OppropBERT RL model. The goal here is for the OppropBERT model to automatically learn a meaningful representation of the type rules and then predict the appropriate annotations for the given input code. The user provides the list of possible types in the type system and the type hierarchy (if one exists). The user can also define preference constraint weights for different locations, e.g., return types, parameter types, etc.

Output: An annotated file that type checks so that it can guarantee a defined property of choice and prevent run time bugs.


OppropBERT - RL Fine tuning
Seed files: Publicly available source code
Type Hierarchy with Preference Constraints
Preprocessing
Generalize variable names
Slot generator and masking annotation to be inferred
CFG generator
Tokenizer and attention mask generator
Data structure to support batching of graphs
Tokenized code
CFGs
Actor-Critic PPO
OppropBERT (pre-trained)
Embeddings created using GraphCodeBERT
Custom loss function
Graph Convolution Layers to represent CFGs
Annotation writer
Candidate annotated source code
Type Checker (Environment)
Source coder parser
AST visitor
Annotation processor
Type rules
Success?
N
Y
Penalty
Reward
Annotated and type-checked source code

```
public class class_1 {
    @Nullable Integer value_1;
    @NonNull Integer variable_2 = Integer.valueOf(22);
    void m1() {
        value_1 = null;
        for (; value_1 == null; ) {
            break;
        }
        @Nullable String var_0 = null;
    }
}
```

Figure 3.2: Architecture diagram of OppropBERT with Reinforcement Learning fine-tuning

Preprocessing: The preprocessing steps are similar to Section 3.1

Actor-Critic PPO: The preprocessed tokenized source code and CFGs are passed into the Actor-Critic Proximal Policy Optimization (PPO) agent. We use OppropBERT (as described in Section 3.1) to serve as the building block for the Actor-Critic agent. OppropBERT encodes the input code and CFGs into a vectorized representation. One can view the agent's actions as a set of predicted annotations at the required locations. The RL agent selects the actions by sampling from a probability distribution and then provides the predicted annotations (actions) at the masked locations to the annotation writer to generate a candidate annotated file to be sent to the environment (i.e., the type checker).

Environment: We use a developer-created type checker to serve as feedback to the Actor-Critic PPO RL agent. The type checker takes as an input a program and outputs a binary output signifying whether the program type checks or not. A successful type-checked program must follow the specified type rules and guarantee the particular property holds to prevent the intended run-time bugs. In the case of Java, the Checker Framework type checker uses the Java compiler's parsed source code and produces an Abstract Syntax Tree (AST) to invoke certain annotation processors and the type checker. The type checker invokes the AST visitor (also known as Type Visitor) to determine the annotation and check the validity of the annotated type according to the specified type rules of the type system we are interested in. An adapter library is created to send the feedback to the agent corresponding to whether the last predicted annotation(s) type checked successfully or not.

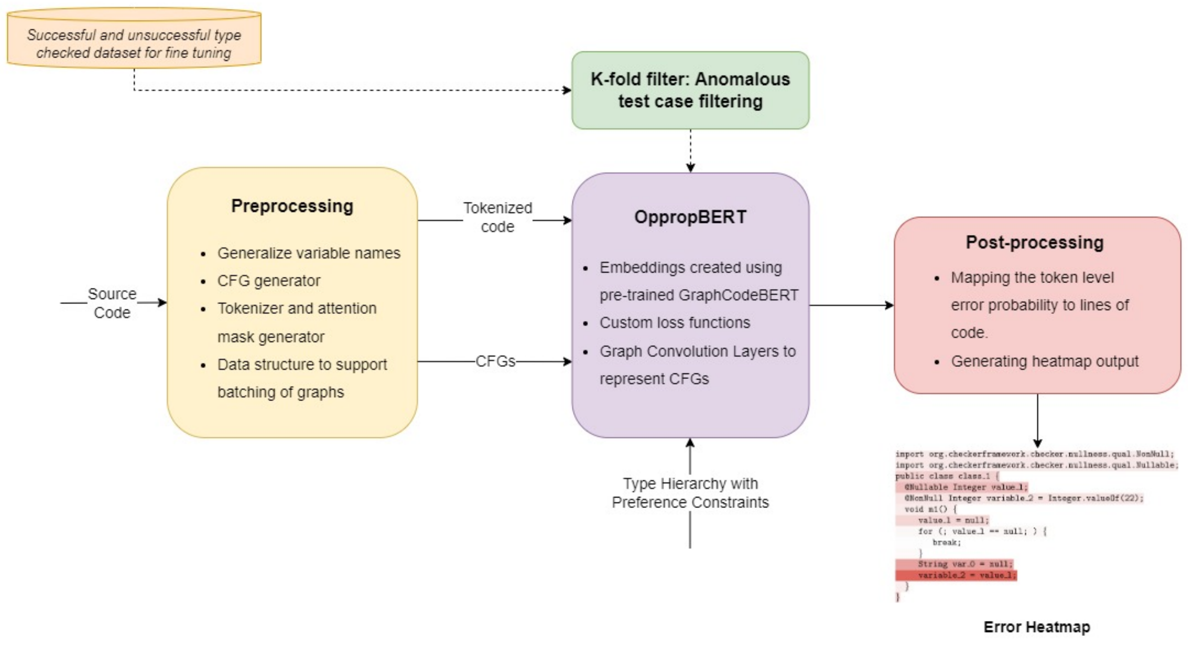
If the type checker supports, the feedback could be fine-grained to the erroneous annotations or lines of code as it would better specify the reward/penalty signals. Using such detailed signals, the RL agent can penalize only the incorrect annotations. On the other hand, a binary signal does not provide any information about individual erroneous locations, penalizing even the correct annotation for certain locations. In this way, the fine-grained feedback can lead to a better reward signal and, thus, faster convergence. Lastly, weak reward signals can also be obtained from a data-driven model trained for predicting type errors in the absence of a developer-provided type checker. More details regarding this can be found in Section 3.3.

Details of Reward and Penalty: In the presence of the fine-grained signal from the type checker, the OppropBERT RL agent (Actor-Critic PPO) receives a vector of reward-penalty signals corresponding to whether the annotations at the respective locations were predicted correctly or not. The vector is transformed into a scalar value using the preference weights if provided by the user, otherwise, it is weighted equally to generate a reward signal. No transformation is required for a binary signal from the type checker, and we can directly use this scalar value as a reward/penalty signal. In the case of a weak signal from a data-driven type checker, the weights can be determined by the confidence of the prediction values. Additional user-defined heuristics as mentioned in Section 3.1 are also incorporated into the reward signals.

Putting it all together: Initially, a file is selected randomly from the list of seed files (publicly available files that can be either annotated or unannotated) that are pre-processed to create tokenized source code and CFGs. The preprocessed input is fed to the pre-trained and fine-tuned (supervised) via an Actor-Critic PPO RL agent. The PPO agent predicts the type-adhering annotations used to generate the candidate annotated file. The candidate annotated file is passed to the type checker (environment) that sends a reward/penalty signal back to the RL agent. Success is detected when the candidate annotated file indeed type checks. In case of a reward (or penalty), the RL agent modifies its probability distribution to prefer (or reject) the annotation for that state. The chosen seed input file is again used for the PPO agent to predict a new candidate annotated file. This loop is repeated a few times before discarding the current seed input file and choosing a new one from the list of input files. The learning process terminates when the desired timeout or epoch has been reached. In the testing phase, a successfully annotated and type-checked file is returned as an output to the user.

Note that the predicting no annotations at a masked location corresponds to the default annotation for that particular location based on how it has been defined in the type checker. These default annotations might not be safe or the best option all the time. The model aims to generate annotations that type checks and also abide by intended user-defined heuristics.

3.3 Error Heatmap Prediction



Successful and unsuccessful type checked dataset for fine tuning
K-fold filter: Anomalous test case filtering
Preprocessing
Generalize variable names
CFG generator
Tokenizer and attention mask generator
Data structure to support batching of graphs
Source Code
Tokenized code
CFGs
OppropBERT
Embeddings created using pre-trained GraphCodeBERT
Custom loss functions
Graph Convolution Layers to represent CFGs
Type Hierarchy with Preference Constraints
Post-processing
Mapping the token level error probability to lines of code.
Generating heatmap output
Error Heatmap

```
import org.checkerframework.checker.nullness.qual.NonNull;
import org.checkerframework.checker.nullness.qual.Nullable;
public class class_1 {
  @Nullable Integer value_1;
  @NonNull Integer variable_2 = Integer.valueOf(22);
  void m1() {
     value_1 = null;
     for (; value_1 == null; ) {
        break;
     }
     String var_0 = null;
     variable_2 = value_1;
  }
}
```

Figure 3.3: Architecture diagram of OppropBERT for error heatmap prediction task

Given an input program, the error heatmap prediction model gives a token-level heatmap, where the heatmap corresponds to the probability of error at different locations in the code (Figure 3.3). Before type annotation prediction, the error heatmap prediction model can be used to make sure that the unannotated or partially annotated code provided by the user type checks or not. Moreover, in the absence of a type checker for the RL loop, as described in the last section, the heatmap prediction model can also serve as a weak signal for certain examples it is confident about.

Input: A file written in a particular programming language to be type-checked is given as the input. The file could either be unannotated or annotated. The user provides the list of possible types in the type system and the type hierarchy (if one exists).

Output: An error heatmap highlighting the locations with a high probability of type error. The absence of errors signifies that the code adheres to the type rules to guarantee with high accuracy that a defined property of choice holds and prevents run-time bugs.

Preprocessing: The preprocessing steps are similar to Section 3.1 except for the annotation mask creation. Heatmap labels for the tokenized test cases are from the given error line numbers by mapping the line numbers to the respective set of tokens.

OppropBERT: The model architecture remains the same as described in Section 3.1.

Post-processing: During the post-processing stage, the predicted token-level error probabilities by OppropBERT are mapped back to the source code to generate the heatmap. The error probability can be token-level or line-level (by aggregating the token-level probabilities for the particular line).

Chapter 4

Implementation

4.1 Comparative tools

For Java's nullness type system dataset, we compare OppropBERT against 4 popular nullness type checking or annotation inference tools (for more details, please refer to Section 2.1):

- Checker Framework [22, 64]
- Spotbugs[1]
- Eclipse[2]
- IntelliJ[3]

We also create the following two baseline models for comparison:

- Genetic Algorithm: Using a fitness function based on the preference constraint to find the best possible set of annotations.

[1]https://spotbugs.github.io/

[2]https://help.eclipse.org/latest/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2Ftasks%2Ftask-using_null_type_annotations.htm

[3]https://www.jetbrains.com/help/idea/inferring-nullity.html

- Random Baseline: Randomly assigning possible type annotations to the available slots.

For the optional type prediction task, we compare against the following 7 benchmarks:

- BERT [21]: Bidirectional Encoder Representations from Transformers model pre-trained using masked language modeling and next sentence prediction tasks.
- RoBERTa [53]: A robustly optimized BERT pre-training approach which is built on top of BERT, but with modified hyperparameters, next-sentence pre-training objective removed, and dynamic masking.
- CodeBERT [25]: A bi-modal pre-trained model for programming language (PL) and natural language (NL) related tasks. The pre-training objectives include masked language modeling and replaced token detection.
- CodeBERTa [89]: It is a RoBERTa-like model pre-trained on the CodeSearchNet dataset [38] from GitHub.
- GraphCodeBERT [29]: An improved version of CodeBERT, where the authors use data flow information in the pre-training stage by introducing two structure-aware pre-training tasks to capture the inherent structure of code.
- PolyGot [1]: CodeBERT model with multi-lingual fine-tuning.
- GraphPolyGot [1]: GraphCodeBERT model with multi-lingual fine-tuning.

4.2 Benchmarks

We collect the Java test cases from Checker Framework nullness type system[4]. We generalize their variable names and augment the Java files by including dummy function calls and statements to increase the length of the program. We also mix and match with the existing codebase to create more combinations. To create additional test cases, we mask all possible slots where an annotation could be inserted and insert all possible combinations of annotations at different slot locations. Since we have a Nullness type checker available

[4]https://github.com/opprop/checker-framework/tree/master/checker/src/main/java/org/checkerframework/checker/nullness

in Checker Framework, we could feed the newly generated test cases to this type checker to verify whether the test case type checks.

Furthermore, we use a deduplication tool popular in the literature [4] based on Jaccard similarity to remove all the duplicate test cases. After augmentation, we have 3000 Java files with 50163 lines of code and, on average, eight annotations in every file. We traverse each file's AST (abstract syntax tree) to extract the annotations. This way, we gather the tokens and their type annotations on the AST nodes to create an aligned pair. If the type annotation does not exist, we query the default annotation processor of the Checker Framework to store the default annotation. The train-validation-test split is 70:10:20. The division is done so that all the augmented files of a particular set of error types remain in the same data split to avoid any leakage.

For testing the models, we use popular Java repositories also used by [22]. Note that all the lines of code were calculated after ignoring the comments and blank lines:

- Daikon library[5] with 117472 lines of Java code.
- Options library[6] with 1915 lines of Java code.
- Lookup library[7] with 274 lines of Java code.

The advantage of using our method is that for migrating to a new type system, one would need to simply create the test cases with all possible combinations of annotations and feed it to the model. Furthermore, they can use the filtering algorithm to find anomalous test cases and double-check the presence of any errors in the test cases.

For the TypeScript type prediction task, we use the largest standard benchmark ManyTypes4TypeScript [39] with over 9 million type annotations, across 13953 projects and 539571 files. ManyTypes4TypeScript is also part of the popular CodeXGLUE [55] benchmarks. Inspired by the growth of diversified benchmark datasets in the field of computer vision and NLP, researchers from Microsoft Research Asia, Developer Division, and Bing introduced a benchmark dataset for code intelligence called CodeXGLUE (General Language Understanding Evaluation benchmark for CODE).

Lastly, for the Python type prediction task, we manually create a return-type warning prediction dataset. A software developer often forgets to store a return value from a method call that modifies the input arguments (e.g., inserting/deleting in a string/NumPy

[5]`https://github.com/codespecs/daikon`
[6]`https://github.com/plume-lib/options`
[7]`https://github.com/plume-lib/lookup`

array/dataframe). Such a mistake usually happens when the user accidentally assumes the method will perform an in-place operation instead of returning the modified object. On the other hand, sometimes a developer might be interested in looking at the intermediate output being printed in the method itself or merely printing the output to test out a functionality (the IPython Notebook allows printing intermediate lines of code). Employing a static analysis tool will generate a lot of false positive warnings because the warning depends on user behavior rather than type rules. Therefore, given a user code, the body of the method in question, the model can also predict whether the developer accidentally missed the return value or intentionally did not store the modified return value.

4.3 Computational Environment

We implemented OppropBERT in over 2000 lines of Python 3 using Pytorch for implementing the Deep Learning model. The code and benchmark would be available on my GitHub repository and my homepage. We performed our experiments on Digital Research Alliance of Canada computing service [8], a CentOS V7 cluster of Intel Silver 4216 Cascade Lake running at 2.10 GHz with 32 GB memory and 4 Nvidia V100 Volta GPUs.

4.4 Metrics

The metrics used by us have been widely used in the field of type prediction [39, 60] that serve as an important indicator for establishing the efficacy of type prediction models. We evaluated the tools on the following criteria:

- **Accuracy (Top-k):** Ratio of correct type predictions to the total types predicted by the model. A variation of this called the top-k accuracy considers that the model predicted the correct type if the actual type (label) appears in one of the top k predictions from the model. Unless specified otherwise, simply saying accuracy implies top-1 accuracy.
- **Precision:** Fraction of relevant instances among the retrieved instances.
- **Recall:** Fraction of retrieved relevant instances.
- **F1 score:** The harmonic mean of precision and recall.

- **Mean Reciprocal Rank (MRR):** The MRR is expressed as

$$MRR = \frac{1}{|N|} \sum_{i=1}^{|N|} RR(i) \tag{4.1}$$

where, N is the total number of prediction queries from the model, i is the predicted type, and RR is the reciprocal rank which is given as:

$$RR(i) = \begin{cases} \frac{1}{|r_i|}, & \text{if correct type annotation is predicted} \\ 0, & \text{otherwise} \end{cases} \tag{4.2}$$

where, r is the rank of the correctly predicted type annotation in the ranked list. Similar to Top-1 accuracy, a score of 1 is given when the Top-1 suggested type is correct for a prediction. We refer to the MRR of Top-k prediction as MRR@k.

Lastly, regarding the heatmap prediction, we use a metric called **Intersection over Union (IoU)** derived from bounding box object detection and recognition models in Computer Vision [69]. IoU is a numeric value between 0 and 1 that would specify the overlap between the predicted and the ground truth error heatmap tokens. An IoU of 0 signifies no overlap, while an IoU of 1 denotes a perfect overlap between the predicted and actual error heatmap tokens.

4.5 Hyperparameters

We use grid search to find suitable hyperparameters by evaluating on the validation set. WANDB was used as the hyperparameter tuning and experiment tracking platform. The hyperparameter settings are given in Table 4.1. We use the AdamW optimizer [54], an efficient version of the Adam optimizer with decoupled weight decay. Similar to the Transformers implementation [83], we used a learning rate scheduler with a linear warmup. To prevent overfitting, we used dropout, label smoothening, and early stopping with patience of 50 epochs on the validation loss.

Hyperparameter	**Value**
Latent Dimension	768
Optimizer	AdamW
Weight Decay	0.01
Learning Rate	2e-5
GCN Aggregation Type	Sum
GCN Combination Type	Concatenation
GCN Layers	2
RL Policy Gamma	0.99
Policy Learning Rate	3e-5
Max Gradient Clipping	0.5

Table 4.1: Hyperparameter settings

4.6 Effective training strategies

Gradient Accumulation

Instead of calculating the gradients for the entire batch at once, we can break it down into smaller steps. To do so, we need to calculate the gradients of smaller batches and iteratively accumulate the gradients during the forward and backward passes. After a certain number of these iterative accumulations, the optimizer step of the model can be performed. In this manner, we can increase the effective batch size even though this effective batch size would not fit in the GPU memory all at once. The tradeoff for a drastic reduction in the memory overhead is a slight increase in the training time. The accumulation steps are chosen in such a way that we can maximize GPU utilization.

Gradient Checkpointing

Gradient Accumulation, as mentioned above, does help in increasing the effective batch size. However, there can be a case for large models where the GPU runs out of memory even for a batch size of 1. This happens because all the activations from the forward pass are saved to calculate the backward pass gradients, which increases the memory footprint. One way to solve the problem is to remove all the activations computed during the forward pass, and instead re-calculate those values on-demand during the backward pass. The drawback of this approach is the significant increase in computational overhead, which would slow down the training process. Gradient checkpointing leverages the tradeoff between memory vs. time to judiciously save a subset of activations in the computational graph so that not all activations need to be re-calculated during the backward pass for the gradients.

Mixed precision training and pinned memory

To increase the computation speed, the Huggingface library [89] offers mixed-precision training where the variables are stored in half (16-bit) or full (32-bit) floating-point precision. However, there still exists a bottleneck. One of the critical requirements to reach great training speed is the ability to feed the GPU at the maximum rate it can handle. By default, everything happens in the main process, and it might not be able to read the data from the disk fast enough, thus creating a bottleneck, leading to GPU under-utilization. To enable faster transfer from the CPU to the GPU memory, the data can be pre-loaded to the pinned memory of the CPU.

Class weights

In case of class imbalance, it is required to introduce class weights during loss calculation so that the model remains unbiased and pays greater attention to the under-represented classes. The class weights are inversely proportional[8] to the distribution (percentage) in the training set.

Distributed training

The training process can be distributed across multiple GPUs to speed up the training process. We adopt the data parallelization process where initially, the model is replicated

[8]https://www.tensorflow.org/tutorials/structured_data/imbalanced_data#calculate_class_weights

across all the GPUs, and each GPU takes its own mini-batch of data. During the backward pass, the local gradients from every GPU are averaged across all the processes.

Chapter 5

Experiment and Results

In this section, we discuss the results of using OppropBERT for error heatmap prediction and annotation prediction tasks in Java, type prediction in TypeScript, and error prediction in Python.

5.1 Error heatmap and annotation prediction in Java

5.1.1 A few examples of test cases

We first look at a few examples of the test cases that were used for fine-tuning. Listing 5.1 shows a code snippet with a null-pointer exception (NPE) on the `var_1.toString();` line outside the loop.

Listing 5.1: Test case with loop containing NPE

```java
public class class_1 {
    void method_1() {
        String var_1 = "m";
        for (var_1 = null; var_1 != null; var_1 = "m") {
            var_1.toString();
        }
        var_1.toString();
    }
}
```

The initial expression of the loop sets the variable `var_1` to null. Since the condition expression is false, the program breaks out of the loop and dereferencing `var_1` outside the loop results in a null pointer exception.

On the other hand, Listing 5.2 shows an example where there is no null pointer exception in the code. The while loop breaks if the variable `var_1` is assigned null so it never gets dereferenced.

Listing 5.2: Test case with loop that is free from NPE

```
public class class_1 {
    void method_1() {
        String var_1 = null;
        while (true) {
            if (var_1 == null) break;
            var_1.toString();
        }
    }
}
```

Our test cases also have pre-condition and post-condition annotations present, as explained in Section 2.1.3. The pre-condition annotation indicates that the method expects the specified expressions to be non-null when the annotated method is invoked. On the other hand, the post-condition annotation indicates that the value expressions are non-null just after a method call if the method terminates successfully.

5.1.2 Ablation studies and comparative analysis

We use the Java files from the test benchmarks and perform an ablation study to observe how every component of the model plays an essential role in improving the predictions.

As shown in Table 5.1, we observe that introducing the Reinforcement Learning (RL) feedback loop leads to the most significant (+4.5%) increase in Top-1 accuracy as the RL loop is focused on predicting the best set of annotations. Using Graph Neural Network (GNN) alone does not lead to a drastic improvement. However, pairing it with a custom structure-aware loss function led to a significant increase in heatmap IoU (+6.9%), top-1 accuracy (+3.4%), and MRR (+5%).

In terms of the time taken, OppropBERT was faster than Eclipse and IntelliJ by 20% for the annotation prediction task. Note that Checker Framework Inference and Spotbugs

Model	Heatmap IoU (Error Prediction)	Top-1 Accuracy (Annotation Prediction)	MRR@5 (Annotation Prediction)
Random	0	0.5%	3.2
GraphCodeBERT (Baseline)	84.8	81.6%	83.0
+ GNN	85.3	82.8%	83.9
+ Custom loss	**90.7**	84.4%	87.2
+ RL feedback	90.2	**88.9%**	**90.1**

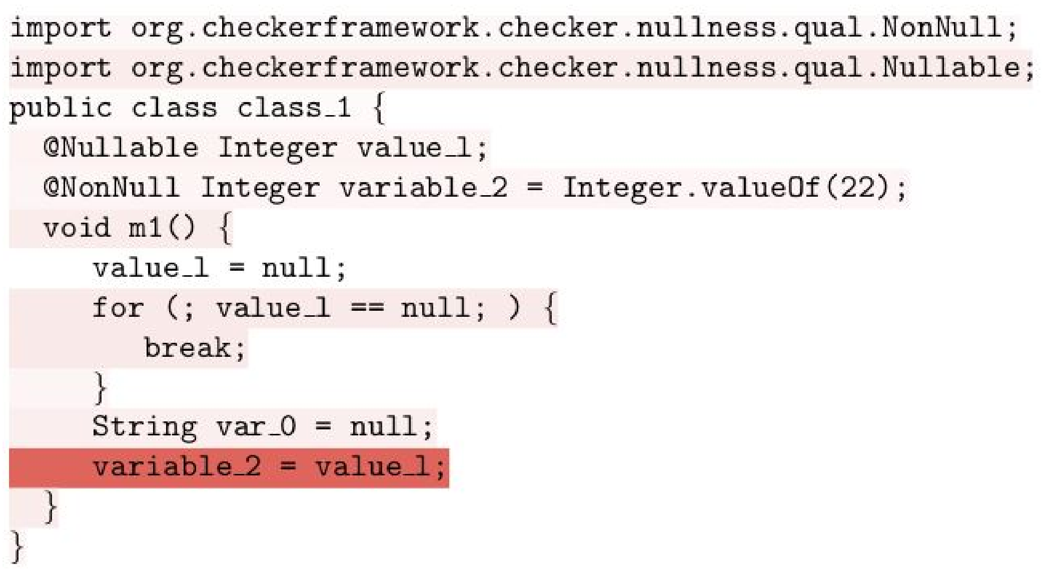
Table 5.1: Impact of design choices: Ablation studies

do not have a nullness type inference system. For the error prediction task, OppropBERT is faster than Checker Framework's nullness type checker by 85%. Figure 5.1 shows an example of an error heatmap prediction output.

Comparison with Genetic Algorithm for preference constraints

Let's start with a brief overview of the Genetic Algorithm. An initial population created at random is the first step of the method. After that, the algorithm generates a series of new populations. The algorithm builds the subsequent population at each stage using members of the current generation. The method evaluates each member of the existing population by calculating its fitness value before generating the new population. These raw fitness scores have now been adjusted to create a more usable range of numbers, otherwise known as expectation values. The parents are the members who are chosen based on their expectations. The elite are some of the more physically fit members of the current population. The subsequent population inherits these elite individuals. The offspring are generated from the parents. Crossover is the process of mixing the vector entries of two parents, while mutation produces random changes to a single parent. The children are then used to create the next generation, replacing the current population. The algorithm terminates when one of the stopping conditions is satisfied.

Recall that the predictions can be made more or less conservative with the help of



```
import org.checkerframework.checker.nullness.qual.NonNull;
import org.checkerframework.checker.nullness.qual.Nullable;
public class class_1 {
  @Nullable Integer value_1;
  @NonNull Integer variable_2 = Integer.valueOf(22);
  void m1() {
     value_1 = null;
     for (; value_1 == null; ) {
        break;
     }
     String var_0 = null;
     variable_2 = value_1;
  }
}
```

Figure 5.1: Example of a heatmap prediction output. The darker shades of red denote a higher probability of error.

preference constraints. For a solver-based approach used by Checker Framework inference, these preference (soft) constraints are considered as breakable constraints when translated to a weighted SAT formula. A weighted MaxSAT solver can be used in such a case to find out a solution such that the hard type system constraints are satisfied and, at the same time, the breakable constraints result in the maximum possible weight. A developer must play around with these weight values and manually analyze the solutions to ensure the strictness they want their heuristic to be enforced. We developed a Genetic Algorithm-based optimization for a MaxSAT solver to find the best soft constraint weights where the fitness function was set with respect to the heuristic the user wants to be enforced (e.g., higher number of `@Rep` annotations for the Universe type system). Given enough iterations (generations), the algorithm is able to converge to a set of "better" weights. However, it takes a lot of time to explore the different sets of weights which involves calling the expensive MaxSAT solver for every chromosome in every generation's population. Moreover, the process is stateless, i.e., the set of weights is decided for a particular program without having any encoding of the program as an input.

A BERT-style model like OppropBERT, on the other hand, can encode the program under consideration along with the user-defined heuristic to suggest the annotations based on the learned structure of the training and fine-tuning examples.

Model	Heatmap IoU (Error Prediction)	Top-1 Accuracy (Annotation Prediction)	MRR@5 (Annotation Prediction)
Zero-shot	42.8	57.3%	61.4
Few-shot	58.0	62.9%	65.5

Table 5.2: Transfer learning on a simple security lattice

5.1.3 Transfer learning to security type system

We use the OppropBERT model trained for nullness type system on a simple security lattice[1]. The security type system has three new annotations that capture only the subtyping relationships and ignores all the other intricate type rules in the nullness type system. In this security lattice, `@TopSecret` is the top annotation, `@Confidential` is the subtype of `@TopSecret`, and `@Public` is the subtype of `@Confidential`. Recall that the nullness type system only has two annotations `@Nullable`, which is the top annotation, and `@NonNull` which is the subtype of `@Nullable`.

It is interesting to note that by replacing the type hierarchy of the nullness type system with the security type system and, thus, preserving the subtyping relationship rules, the model can predict error heatmap and annotations with reasonable accuracy (Table 5.2) even under zero-shot conditions (no training examples of the new type system are provided). This is possible because OppropBERT can capture the understanding of the subtyping relationships into the embedding representation of the annotations of the previous type system and transfer them to the new type system. The model struggles the most to accommodate for the third annotation being introduced, as the last system type only had two. Moreover, fine-tuning the model with some additional examples (few-shot) helps the model to predict slightly better, as shown in Table 5.2.

[1]https://github.com/opprop/security-demo

Listing 5.3: Security lattice example

```
import security.qual.TopSecret;
import security.qual.Confidential;
import security.qual.Public;

public class TypeCheck {

  void topSecretFunc(@TopSecret String p) {}
  void confidentialFunc(@Confidential String p) {}
  void publicFunc(@Public String p) {}

  void parameter(@TopSecret String s1,
                 @Confidential String s2,
                 @Public String s3) {

    topSecretFunc(s1);    // OK
    publicFunc(s1);       // not allowed

  }
}
```

Listing 5.3 shows a simple example to demonstrate the security type system. The `topSecretFunc()` method only accepts a string of type `@TopSecret` or any subtype of it. That is why the function call `topSecretFunc(s1);` is allowed. On the other hand, `publicFunc()` accepts a string of type `@Public` or any subtype of it. Since the type of `s1`, i.e., `@TopSecret` is not a subtype of `@Public`, the function call `publicFunc(s1);` is forbidden and results in an error.

5.1.4 False positive/negative in other tools

Listing 5.4 shows a false negative in Spotbugs nullness type checker, i.e., Spotbugs does not report the error for the assignment `this.obj2 = this.obj1;` on the last line of `bar()` method. The assignment is not possible because the field `obj2` is of type `@NonNull` but is assigned a null object. OppropBERT does report an error for the program correctly.

Listing 5.4: Spotbugs issue

```
public class NullnessExample {

    @Nullable Object obj1;
    @NonNull Object obj2;

    public void foo() {

        this.obj1 = new Object();
        this.bar();

    }

    public void bar() {

        this.obj1 = null;
        this.obj2 = this.obj1; // no error raised

    }
}
```

Listing 5.5 shows a false positive in the Checker Framework nullness type checker. The three `if` conditions consider all the possible scenarios where either `s1` or `s2` could be null and then ends with a return statement for the respective then-branches. Therefore, the last two dereferences of `s1.toString()` and `s2.toString()` must be safe as both are guaranteed to be `@NonNull`. However, Checker Framework's nullness checker issues errors on these two lines, while OppropBERT does not raise any errors.

It would be interesting to see as a possible future work whether the Deep Learning model is indeed able to understand the logical statements or if it is merely pattern matching with some standard structure of if-else branches it saw in training set during the learning process. One might argue that due to the use of CFG to represent the program, the CFG nodes could capture the logical aspect and help the model make appropriate decisions.

Listing 5.5: Checker Framework issue

```
public class NullnessExample {

    public static void foo() {

        String s1 = null;
        String s2 = null;

        if (s1 == null && s2 == null) {
            return;
        }
        if (s1 != null && s2 == null) {
            System.out.println(s1.toString());
            return;
        }
        if (s1 == null && s2 != null) {
            System.out.println(s2.toString());
            return;
        }
        System.out.println(s1.toString());
        System.out.println(s2.toString());
    }
}
```

5.2 Type prediction in TypeScript

For the TypeScript type prediction task, we use the largest standard benchmark Many-Types4TypeScript [39] (also part of the popular CodeXGLUE [55] benchmarks) with over 9 million type annotations, across 13953 projects and 539571 files. We find that OppropBERT outperforms all the other state-of-the-art models previously used for predicting type annotations for TypeScript (Table 5.3). Therefore, we can say that our model can be easily extended to another programming language (e.g., TypeScript) due to the ease of pre-training and fine-tuning.

Model	**Top 100**			
	Precision	**Recall**	**F1 score**	**Accuracy**
BERT [21]	80.04	81.5	80.76	84.97
CodeBERTa [89]	81.31	82.72	82.01	85.94
RoBERTa [53]	82.03	83.81	82.91	86.25
GraphPolyGot [1]	83.80	85.23	84.51	87.40
PolyGot [1]	84.45	85.54	84.95	87.72
CodeBERT [25]	84.58	85.98	85.27	87.94
GraphCodeBERT [29]	84.67	86.41	85.53	88.08
OppropBERT (ours)	**85.22**	**86.90**	**86.05**	**88.39**

Table 5.3: Accuracy Comparisons On ManyTypes4TypeScript

5.3 Error prediction in Python

We use OppropBERT to predict return-type warnings in Python. A software developer often forgets to store a return value from a method call that modifies the input arguments (e.g., inserting/deleting in a string/NumPy array/dataframe). Such a mistake usually happens when the user accidentally assumes the method will perform an in-place operation instead of returning the modified object. For instance, listing 5.6 demonstrates that the user forgets to store the modified dataframe returned by `df.drop_duplicates()` before calling the `df.head()` method to view the modified dataframe. The corrected version of Listing 5.6 is shown in Listing 5.7.

Listing 5.6: Error

```
##%
import pandas as pd

df = pd.DataFrame({
'club': ['WUSA', 'GSA', 'Jam␣Network', 'DSA', 'Fencing'],
'category': ['A', 'A', 'P', 'P', 'S'],
'rating': [4, 4, 4.5, 5, 4.5]
})

df.drop_duplicates() # not an inplace operation
df.head()
```

Listing 5.7: No Error

```
##%
import pandas as pd

df = pd.DataFrame({
'club': ['WUSA', 'GSA', 'Jam␣Network', 'DSA', 'Fencing'],
'category': ['A', 'A', 'P', 'P', 'S'],
'rating': [4, 4, 4.5, 5, 4.5]
})

df = df.drop_duplicates() # OK
df.head()
```

On the other hand, sometimes a developer might be interested in looking at the intermediate output being printed in the method itself or merely printing the output to test out a functionality, as shown in Listing 5.8 (the IPython Notebook allows printing intermediate lines of code). Employing a static analysis tool will generate a lot of false positive warnings because the warning depends on user behavior rather than type rules. Therefore, given a user code, the body of the method in question, the model can also predict whether the developer accidentally missed the return value or intentionally did not store the modified return value.

Listing 5.8: No error

```
##%
import pandas as pd

df = pd.DataFrame({
'club': ['WUSA', 'GSA', 'Jam␣Network', 'DSA', 'Fencing'],
'category': ['A', 'A', 'P', 'P', 'S'],
'rating': [4, 4, 4.5, 5, 4.5]
})

##%
df.drop_duplicates() # OK
```

We created a dataset with ten commonly used functions in Pandas, a popular Python library used for data manipulation and analysis with the help of data frames. We trained the OppropBERT model to predict the return-type warnings with 98% accuracy. A developer can easily extend the model to handle user-defined functions and other libraries by feeding in the function definition and the program-under-test to predict the warnings using the model.

Chapter 6

Related Work

6.1 Natural Language Processing (NLP) for code understanding tasks

In software engineering, there has been a surge of Machine Learning and other statistical approaches for complex code understanding tasks in recent years. Inspired by the success of large pre-trained models on natural language processing (NLP) [9, 21, 77], large sets of programming language data have been used to create pre-trained models such as IntelliCode [81], and GraphCodeBERT [29]. CodeXGLUE [55] is a popular benchmarks that provides such programming language data for code understanding and generation tasks across different modalities, such as, code-code (e.g., clone detection, code completion, code repair, translation), text-code (e.g., natural language code search), code-text (code summarization), and text-text (documentation translation).

Many research groups have leveraged the availability of open-source code on GitHub to solve various tasks using large-scale Deep Learning techniques. One of the recent popular ones is AlphaCode [49] developed by OpenAI. It uses a generative model created to solve competitive coding problems. It was pre-trained on a large database of open-source code available on GitHub and fine-tuned on solutions obtained from CodeContest. Given an unseen programming problem during inference, the model generates novel solutions that perform at the median competitive programmer's level.

Researchers started using standard NLP techniques to solve various code understanding problems by applying them to raw code. DeepTyper [31] used a Bi-directional Gated Recurrent Units (Bi-GRU) model on raw code to predict Javascript and Typescript type assignments. Later on, manual feature selection using only a hand-picked subset of the raw code gained some popularity. NL2Type [56] used a standard Long short-term memory (LSTM)-based classification model to infer Javascript Function Types. It only considers the raw code's comments, function names, and parameter names to predict the types. NLP-based models have also been used for software defect prediction. [13, 48, 67, 85, 86]

6.2 Abstract Syntax Tree (AST)

As an Abstract Syntax Tree (AST) better captures the syntactical structure of the source code, researchers eventually started using them along with the available raw code. One of the earliest works of using Deep Learning with AST for vector representation of programs is the work by Peng et al. [65] where they proposed a "coding criteria" to create learned function, statement, and AST-node-level vector representations. Their idea of creating node-level representation was similar to the node-updation rules of recently popularized GCN models. The current node vector is a combination of its child node(s) passed through a feed-forward layer. These representations were used to solve program classification tasks. ASTs have also been used to create formal program specifications. C2S [93] uses the function's natural language comments, and an intermediate representation IR translator is used to generalize a text-based JML specification to a parsed AST to create specifications, improving the static taint analysis. Code2Seq [6] encodes an AST using a bidirectional Long Short-Term Memory (Bi-LSTM) network. It encodes by concatenating all the paths and the respective tokens. Using such a syntactic encoding of code, they display clear effectiveness of this compositional encoding by treating, let's say, a 'For' and a 'While' loop identically.

DeepAnna [52] uses another commonly used recurrent model, Bi-directional Gated Recurrent Units (Bi-GRU), for annotation recommendation for popular Java frameworks, such as Spring and Hibernate. DeepAnna is meant for developers who often need help determining the correct annotations to use due to many similar-behaving annotations defined by these frameworks. Rather than being an annotation recommender, OppropBERT is meant for inserting annotations in the code to prevent runtime errors. DeepAnna uses an AST traversal algorithm to convert the AST into a sequential representation. Moreover, the DeepAnna annotation prediction task only predicts annotation at the class and method level, where they would only have to predict one annotation per input code snippet. In contrast, OppropBERT predicts for all possible annotation locations, such as fields, parameters, local variables, etc., where multiple annotations need to be predicted for a given input program. Also, using an N-independent binary classification approach makes the

DeepAnna inference slow during training and testing. All these techniques do not use the graphical nature of the ASTs and instead use a simplified sequential transformation of the same. OppropBERT, on the other hand, uses the graphical structure of the Control Flow Graph (CFG) using a Graph Neural Network (GNN) along with a BERT-style model making it more structure-aware.

To handle the open-vocabulary issue (certain user-defined types not available on the training set but appear in the test set) of predicting type annotations, a representation learning combined with a clustering approach was proposed by Type4Py [60]. They use code context and visible type hints extracted from the AST to infer optional type annotations, introduced by PEP 484 in Python. Since there can be many possible type annotations (built-in and user-defined), creating a classification model with a certain number of classes won't be able to generalize well. Instead of treating it as a classification problem, Type4Py assigns the nearest type clusters (handling one annotation at a time) learned during training. On the other hand, Typilus [3] adopted a k-Nearest Neighbours approach on top of the Graph neural networks (GNN)-based encoding created by integrating information from identifiers, syntactical structure, and dataflow to infer type annotations for Python. However, it requires a sophisticated source code analysis to create its graph representations.

6.3 Control Flow Graph (CFG)

A Control Flow Graph (CFG) provides a graphical representation of a program's control flow during its execution. So, for the tasks that rely more on the program's execution flow than its syntactical structure, incorporating CFGs as input to the model seems to improve the results. Convolutional Neural Networks (CNNs) have been used as a popular choice to represent AST and CFG. Phan et al. [66] used CNNs to better predict software defects by capturing the appropriate semantics of the given assembly code. Mou et al. [62] extended the idea of a CNN to tree-based inputs to identify specific code patterns from an AST representation of the input code. Liang et al. [51] used both AST and CFG to detect malicious JavaScript using CNNs, capturing both the abstract syntactic structures and execution processes.

6.4 Transformers and pre-trained models

Transformer model [83] was a breakthrough in the NLP domain because of a faster parallelized multi-head self-attention approach that provided better feature representations than their recurrent predecessors. Bi-directional Encoder Representations from Transformers (BERT) [21, 53] architectures, as the name suggests, make use of Transformers along with valuable pre-training objectives to improve the encoder representations. The increased popularity of BERT-style models soon paved its path in the software engineering community. Mashhadi and Hemmati [57] used raw code and a pre-trained BERT-style model to find simple bugs in Java from a curated dataset. TypeBERT [40] predicts type annotations for Typescript language using a BERT-style model with raw code as input outperforming the previous state-of-the-art LambdaNet [88]. They found that given enough data, we do not need sophisticated inductive biases to develop a type prediction model.

The use of creative pre-processing and pre-training objectives focused on the programming languages under consideration have also been seen to help improve the models to generalize across different tasks as they can create better feature vector representations for the code. CodeBERT [25] is one such encoder-only bi-modal model for better Natural Language (comments) - Programming Language representation. It includes the Replaced Token Detection (RTD) pre-training objective apart from the Masked Language Modeling (MLM). CodeBERT makes use of WordPiece to tokenize the code blocks. WordPiece works by finding a symbol pair whose probability, when divided by the product of the probabilities of the two symbols separately, is the greatest among all the available symbol pairs. In this way, unlike Byte-Pair Encoding (BPE), WordPiece makes sure that merging the two symbols is worth it by evaluating the loss from the merge. GraphCodeBERT [29], unlike its predecessor CodeBERT, considers Abstract Syntax Tree (AST) to gather variable relation to create a dataflow graph. The model can now use this dataflow graph effectively by utilizing the node alignment and edge prediction pre-training objectives. For the next iteration, the authors developed UniXCoder [28] to create a unified cross-modal pre-trained model which works in an encoder-only, decoder-only, and encoder-decoder setting. Similarly, researchers at Facebook AI Research developed several iterations of an encoder-decoder model [44, 70, 71] for translating between programming languages making use of different pre-training objectives, e.g., cross-lingual MLM, denoising, keyword-focusing masked language modeling, and unit-test verified back translation. However, none of these models leverage the graph representation of AST or CFG using a Graph Neural Network-based approach.

6.5 Graph Neural Networks (GNNs)

Graph neural networks (GNNs) have gained much popularity over the past few years. Deep Learning models for software engineering have started using GNNs to solve programming language tasks. One of the earliest works using GNNs is the gated graph neural network (GGNN) [50], where the authors extended the GNNs by adding GRU layers [17] for updating the hidden states of the graph nodes to solve a program verification task by approximating reachable program states. Later on, Allamanis et al. [4] used GNNs for predicting variable names and selecting the correct variables by using the graphs derived from ASTs for the GNN-based model outperforming the recurrent neural network models that merely used raw source code. One of the popular GNN models, Graph Convolution Network (GCN), has also been used for malware classification problems. Previously, the researchers used to rely a lot on handcrafted rules and features. However, authors in [92] have found that representing a program as a CFG and making use of a GCN does help to classify malware better. Hua et al. [34] converted assembly code call instructions to CFG and used GCN to classify malware. FUNDED [84] used graph-based models for code vulnerability detection. ShadowGNN [15] combines both the natural language and SQL schema in a GCN followed by a Transformer structure to create a Text-to-SQL parser, improving the generalization capability of the model on rare and unseen schema. Dinella et al. [24] used a graph-based representation of Javascript programs to predict the position of buggy nodes and the required graph modifications to fix them. LambdaNet [88] creates a type dependency graph and then uses a GNN-based approach along with some handcrafted logical constraints and contextual hints (e.g., variables assignments and names) for predicting types for TypeScript programs. Researchers have also started to use GNNs for other code understanding tasks such as program similarity [63], software vulnerability detection [16,94], code completion [20], and code summarization [46]. For a comprehensive overview of GNNs, we refer the readers to Wu et al. [90], where they provide a thorough review and comparisons of different GNN models. Most GNN models rely on handcrafted feature extraction, limiting their generalization capabilities to different tasks. Furthermore, they do not leverage an unsupervised learning approach with pre-training objectives focused on the graphical representation of the source code.

There has been no previous work for predicting type annotations for Java programs to prevent run-time bugs using a data-driven approach using a structure-aware custom loss function, transfer learning to a new type system with fewer training data, or using a Reinforcement Learning (RL)-based approach for additional fine-tuning of a supervised annotation prediction model.

Chapter 7

Conclusion and Future Work

In this thesis, we present OppropBERT, a novel GNN and BERT-style RL-based type inference system. OppropBERT is the first Machine Learning based type inference system that uses developer-written test cases and an RL feedback loop without requiring explicit and manually crafted type inference rules. Via an extensive empirical evaluation against state-of-the-art models and tools, we show that our model OppropBERT is automatic, extensible, type-adhering, and efficient. Our model will be most helpful to developers who are developing a type checker or a type inference tool from an available type checker, as the traditional process is time-consuming, error-prone, and expensive. Using OppropBERT, the user does not need to explicitly encode the type inference rules or write a specific type inference framework for every type system. The model learns the type rules implicitly using a data-driven approach. We also define a new token-level error heatmap and annotation prediction task for Java where OppropBERT demonstrates zero and few-shot transfer learning capabilities for other type systems. Furthermore, we show that the model is also extensible to other programming languages, such as TypeScript and Python.

In future work, the model can be extended with a decoder to solve code repair tasks. Code repair is required for programs that cannot be corrected by simply inserting the annotation but require additional repairs in the code structure. Additional information from Javadoc or comments can also be incorporated into the model to make better predictions.

Let us consider Listing 7.1 for the security type system. `@Public` is the default annotation in the hierarchy. The method call `method2(s2);` is invalid because `method2` expects a `@Public` string. The inference framework can easily fix this by changing the method signature to `void method2(@TopSecret String p)`.

Listing 7.1: Fixable issues using the inference tool

```
import security.qual.TopSecret;
import security.qual.Confidential;
import security.qual.Public;

public class Parameter {

  void method1(String p) {}
  void method2(String p) {}

  void parameter(@Public String s1, @TopSecret String s2) {
    method1(s1);
    // :: fixable-error: (argument.type.incompatible)
    method2(s2);
  }
}
```

However, if we look at the example in Listing 7.2, the only way to prevent a null-pointer exception is by enclosing the `var_1.toString();` dereferencing under an if-block which checks if `var_1` is not null. Such cases cannot be resolved by using an inference tool. Instead, we need code repair models as mentioned above.

Listing 7.2: Example that the inference tool cannot fix

```
public class class_1 {
    void method_1() {
        String var_1 = null;
        .
        // some code block
        .
        var_1.toString();
        }
    }
}
```

Similarly, for Listing 5.5, it would be interesting to see how well the ML-based inference framework can understand logical statements. Handling penalties and UNSAT cases based on different types of errors for inference or classifying different error types, as mentioned in Section 2.1.3, during error heatmap type checking would be another extension worth exploring in the future. Moreover, the model can also be extended to type systems with a complex representation of types, such as expressive units of measurement types [91] or more expressive type systems, such as Property types [45].

References

[1] Toufique Ahmed and Premkumar Devanbu. Multilingual training for software engineering. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1443–1455, 2022.

[2] Miltiadis Allamanis. The adverse effects of code duplication in machine learning models of code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 143–153, 2019.

[3] Miltiadis Allamanis, Earl T Barr, Soline Ducousso, and Zheng Gao. Typilus: Neural type hints. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 91–105, 2020.

[4] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. In *International Conference on Learning Representations*, 2018.

[5] Miltiadis Allamanis and Charles Sutton. Mining source code repositories at massive scale using language modeling. In *2013 10th working conference on mining software repositories (MSR)*, pages 207–216. IEEE, 2013.

[6] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. Code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400*, 2018.

[7] Cyrille Artho. Finding faults in multi-threaded programs. Master's thesis, ETH Zurich, 2001.

[8] Susan Baldwin. Compute Canada: Advancing computational research. In *Journal of Physics: Conference Series*, volume 341, page 012001. IOP Publishing, 2012.

[9] Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S. Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.

[10] Gilad Bracha. Pluggable type systems. In *OOPSLA Workshop on Revival of Dynamic Languages*, 2004.

[11] SRK Branavan, David Silver, and Regina Barzilay. Learning to win by reading manuals in a monte-carlo framework. *Journal of Artificial Intelligence Research*, 43:661–704, 2012.

[12] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[13] Deyu Chen, Xiang Chen, Hao Li, Junfeng Xie, and Yanzhou Mu. Deepcpdp: Deep learning based cross-project defect prediction. *IEEE Access*, 7:184832–184848, 2019.

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

[15] Zhi Chen, Lu Chen, Yanbin Zhao, Ruisheng Cao, Zihan Xu, Su Zhu, and Kai Yu. Shadowgnn: Graph projection neural network for text-to-sql parser. *arXiv preprint arXiv:2104.04689*, 2021.

[16] Xiao Cheng, Haoyu Wang, Jiayi Hua, Guoai Xu, and Yulei Sui. Deepwukong: Statically detecting software vulnerabilities using deep graph neural network. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(3):1–33, 2021.

[17] Kyunghyun Cho, Bart Van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*, 2014.

[18] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.

[19] Tom Copeland. *PMD applied*, volume 10. Centennial Books, San Francisco, 2005.

[20] Milan Cvitkovic, Badal Singh, and Animashree Anandkumar. Open vocabulary learning on source code with a graph-structured cache. In *International Conference on Machine Learning*, pages 1475–1485. PMLR, 2019.

[21] 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.

[22] Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Kivanç Muşlu, and Todd W. Schiller. Building and using pluggable type-checkers. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 681–690, 2011.

[23] Werner Dietl, Michael D. Ernst, and Peter Müller. Tunable static inference for Generic Universe Types. In *ECOOP 2011 — Object-Oriented Programming, 25th European Conference*, pages 333–357, Lancaster, UK, July 2011.

[24] 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.

[25] 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.

[26] Michael Furr, Jong-hoon An, Jeffrey S Foster, and Michael Hicks. Static type inference for ruby. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1859–1866, 2009.

[27] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[28] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850*, 2022.

[29] 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.

[30] Hongyu Guo. Generating text with deep reinforcement learning. *arXiv preprint arXiv:1510.09202*, 2015.

[31] 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.

[32] David Hovemeyer and William Pugh. Finding bugs is easy. In *Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 132–136, Vancouver, BC, Canada, October 2004.

[33] David Hovemeyer, Jaime Spacco, and William Pugh. Evaluating and tuning a static analysis to find null pointer bugs. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 13–19, 2005.

[34] Yakang Hua, Yuanzheng Du, and Dongzhi He. Classifying packed malware represented as control flow graphs using deep graph convolutional neural network. In *2020 International Conference on Computer Engineering and Application (ICCEA)*, pages 254–258. IEEE, 2020.

[35] Wei Huang, Werner Dietl, Ana Milanova, and Michael D. Ernst. Inference and checking of object ownership. In *ECOOP 2012 — Object-Oriented Programming, 26th European Conference*, pages 181–206, Beijing, China, June 2012.

[36] Wei Huang, Ana Milanova, Werner Dietl, and Michael D. Ernst. Reim & reiminfer: Checking and inference of reference immutability and method purity. *ACM SIGPLAN Notices*, 47(10):879–896, 2012.

[37] Marianne Huchard, Christian Kästner, and Gordon Fraser. Proceedings of the 33rd acm/ieee international conference on automated software engineering (ase 2018). In *ASE: Automated Software Engineering*. ACM Press, 2018.

[38] 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.

[39] Kevin Jesse and Premkumar T. Devanbu. ManyTypes4TypeScript: A comprehensive TypeScript dataset for sequence-based type inference. In *Proceedings of the 19th*

International Conference on Mining Software Repositories, MSR '22, page 294–298, New York, NY, USA, 2022. Association for Computing Machinery.

[40] Kevin Jesse, Premkumar T. Devanbu, and Toufique Ahmed. Learning type annotation: is big data enough? In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1483–1486, 2021.

[41] Nahid Juma, Werner Dietl, and Mahesh Tripunitara. A computational complexity analysis of tunable type inference for generic universe types. *Theoretical Computer Science*, 814:189–209, 2020.

[42] Sebastian Kleinschmager, Romain Robbes, Andreas Stefik, Stefan Hanenberg, and Eric Tanter. Do static type systems improve the maintainability of software systems? an empirical study. In *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, pages 153–162. IEEE, 2012.

[43] Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. *arXiv preprint arXiv:2006.03511*, 2020.

[44] Marie-Anne Lachaux, Baptiste Roziere, Marc Szafraniec, and Guillaume Lample. Dobf: A deobfuscation pre-training objective for programming languages. *Advances in Neural Information Processing Systems*, 34:14967–14979, 2021.

[45] Florian Lanzinger, Alexander Weigl, Mattias Ulbrich, and Werner Dietl. Scalability and precision by combining expressive type systems and deductive verification. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–29, 2021.

[46] Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. Improved code summarization via a graph neural network. In *Proceedings of the 28th international conference on program comprehension*, pages 184–195, 2020.

[47] Jason Lee, Kyunghyun Cho, and Thomas Hofmann. Fully character-level neural machine translation without explicit segmentation. *Transactions of the Association for Computational Linguistics*, 5:365–378, 2017.

[48] Jian Li, Pinjia He, Jieming Zhu, and Michael R. Lyu. Software defect prediction via convolutional neural network. In *2017 IEEE international conference on software quality, reliability and security (QRS)*, pages 318–328. IEEE, 2017.

[49] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814*, 2022.

[50] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.

[51] Hongliang Liang, Yuxing Yang, Lu Sun, and Lin Jiang. Jsac: A novel framework to detect malicious javascript via cnns over ast and cfg. In *2019 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2019.

[52] Yi Liu, Yadong Yan, Chaofeng Sha, Xin Peng, Bihuan Chen, and Chong Wang. Deepanna: Deep learning based java annotation recommendation and misuse detection. In *2022 29th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE*, 2022.

[53] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[54] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2019.

[55] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021.

[56] Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. Nl2type: inferring javascript function types from natural language information. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 304–315. IEEE, 2019.

[57] Ehsan Mashhadi and Hadi Hemmati. Applying codebert for automated program repair of java simple bugs. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 505–509. IEEE, 2021.

[58] Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. Studying the usage of text-to-text transfer transformer to support code-related tasks. In *2021 IEEE/ACM*

43rd International Conference on Software Engineering (ICSE), pages 336–347. IEEE, 2021.

[59] Alex McPeak. What's the True Cost of a Software Bug? https://smartbear.com/blog/software-bug-cost/, 2017. [Online; accessed 07-October-2022].

[60] Amir M Mir, Evaldas Latoškinas, Sebastian Proksch, and Georgios Gousios. Type4py: practical deep similarity learning-based type inference for python. In *Proceedings of the 44th International Conference on Software Engineering*, pages 2241–2252, 2022.

[61] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[62] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. In *Thirtieth AAAI conference on artificial intelligence*, 2016.

[63] Aravind Nair, Avijit Roy, and Karl Meinke. funcgnn: A graph neural network approach to program similarity. In *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11, 2020.

[64] Matthew M Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. Practical pluggable types for java. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 201–212, 2008.

[65] Hao Peng, Lili Mou, Ge Li, Yuxuan Liu, Lu Zhang, and Zhi Jin. Building program vector representations for deep learning. In *International conference on knowledge science, engineering and management*, pages 547–553. Springer, 2015.

[66] Anh Viet Phan, Minh Le Nguyen, and Lam Thu Bui. Convolutional neural networks over control flow graphs for software defect prediction. In *2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 45–52. IEEE, 2017.

[67] Lei Qiao, Xuesong Li, Qasim Umer, and Ping Guo. Deep learning based software defect prediction. *Neurocomputing*, 385:100–110, 2020.

[68] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[69] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[70] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. *Advances in Neural Information Processing Systems*, 33:20601–20611, 2020.

[71] Baptiste Roziere, Jie M Zhang, Francois Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. Leveraging automated unit tests for unsupervised code translation. *arXiv preprint arXiv:2110.06773*, 2021.

[72] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897. PMLR, 2015.

[73] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[74] Mike Schuster and Kaisuke Nakajima. Japanese and korean voice search. In *2012 IEEE international conference on acoustics, speech and signal processing (ICASSP)*, pages 5149–5152. IEEE, 2012.

[75] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*, 2015.

[76] David Silver, Richard S. Sutton, and Martin Müller. Reinforcement learning of local shape in the game of go. In *IJCAI*, volume 7, pages 1053–1058, 2007.

[77] Irene Solaiman, Miles Brundage, Jack Clark, Amanda Askell, Ariel Herbert-Voss, Jeff Wu, Alec Radford, Gretchen Krueger, Jong Wook Kim, Sarah Kreps, et al. Release strategies and the social impacts of language models. *arXiv preprint arXiv:1908.09203*, 2019.

[78] Martina Stojmanovska. 10 Biggest Software Bugs and Tech Fails of 2021. https://www.testdevlab.com/, 2021. [Online; accessed 07-October-2022].

[79] Richard S. Sutton and Andrew G. Barto. Reinforcement learning: An introduction. *Robotica*, 17(2):229–235, 1999.

[80] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: An introduction.* MIT press, 2018.

[81] 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.

[82] Harold Thimbleby. Inside medical software: When programming errors cost lives. *ITNOW*, 60(2):50–52, 2018.

[83] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[84] Huanting Wang, Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Yansong Feng, Lizhong Bian, and Zheng Wang. Combining graph-based learning with automated data collection for code vulnerability detection. *IEEE Transactions on Information Forensics and Security*, 16:1943–1958, 2020.

[85] Song Wang, Taiyue Liu, Jaechang Nam, and Lin Tan. Deep semantic feature learning for software defect prediction. *IEEE Transactions on Software Engineering*, 46(12):1267–1293, 2018.

[86] Song Wang, Taiyue Liu, and Lin Tan. Automatically learning semantic features for defect prediction. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 297–308. IEEE, 2016.

[87] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[88] Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. Lambdanet: Probabilistic type inference using graph neural networks. *arXiv preprint arXiv:2005.02161*, 2020.

[89] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*, 2019.

[90] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems*, 32(1):4–24, 2020.

[91] Tongtong Xiang, Jeff Y Luo, and Werner Dietl. Precise inference of expressive units of measurement types. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–28, 2020.

[92] Jiaqi Yan, Guanhua Yan, and Dong Jin. Classifying malware represented as control flow graphs using deep graph convolutional neural network. In *2019 49th annual IEEE/IFIP international conference on dependable systems and networks (DSN)*, pages 52–63. IEEE, 2019.

[93] Juan Zhai, Yu Shi, Minxue Pan, Guian Zhou, Yongxiang Liu, Chunrong Fang, Shiqing Ma, Lin Tan, and Xiangyu Zhang. C2s: translating natural language comments to formal program specifications. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 25–37, 2020.

[94] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems*, 32, 2019.