Chapter 22
Graph Neural Networks in Program Analysis

Miltiadis Allamanis

Abstract Program analysis aims to determine if a program's behavior complies with some specification. Commonly, program analyses need to be defined and tuned by humans. This is a costly process. Recently, machine learning methods have shown promise for probabilistically realizing a wide range of program analyses. Given the structured nature of programs, and the commonality of graph representations in program analysis, graph neural networks (GNN) offer an elegant way to represent, learn, and reason about programs and are commonly used in machine learning-based program analyses. This chapter discusses the use of GNNs for program analysis, highlighting two practical use cases: variable misuse detection and type inference.

22.1 Introduction

Program analysis is a widely studied area in programming language research that has been an active and lively research domain for decades with many fruitful results. The goal of program analysis is to determine properties of a program with regards to its behavior (Nielson et al, 2015). Traditionally analysis methods aim to provide formal guarantees about some program property e.g., that the output of a function always satisfies some condition, or that a program will always terminate. To provide those guarantees, traditional program analysis relies on rigorous mathematical methods that can deterministically and conclusively prove or disprove a formal statement about a program's behavior.

However, these methods cannot learn to employ coding patterns or probabilistically handle ambiguous information that is abundant in real-life code and is widely used by coders. For example, when a software engineer encounters a variable named

Miltiadis Allamanis
Microsoft Research, e-mail: miallama@microsoft.com

"counter", without any additional context, she/he will conclude with high probability that this variable is a non-negative integer that enumerates some elements or events. In contrast, a formal program analysis method — having no additional context — will conservatively conclude that "counter" may contain any value.

Machine learning-based program analysis (Section 22.2) aims to provide this human-like ability to learn to reason over ambiguous and partial information at the cost of foregoing the ability to provide (absolute) guarantees. Instead, through learning common coding patterns, such as naming conventions and syntactic idioms, these methods can offer (probabilistic) evidence about aspects of the behavior of a program. This is not to say that machine learning makes traditional program analyses redundant. Instead, machine learning provides a useful weapon in the arsenal of program analysis methodologies.

Graph representations of programs play a central role in program analysis and allow reasoning over the complex structure of programs. Section 22.3 illustrates one such graph representation which we use throughout this and discusses alternatives. We then discuss GNNs which have found a natural fit for machine learning-based program analyses and relate them to other machine learning models (Section 22.4). GNNs allow us to represent, learn, and reason over programs elegantly by integrating the rich, deterministic relationships among program entities with the ability to learn over ambiguous coding patterns. In this , we discuss how to approach two practical static program analyses using GNNs: bug detection (Section 22.5), and probabilistic type inference (Section 22.6). We conclude this (Section 22.7) discussing open challenges and promising new areas of research in the area.

22.2 Machine Learning in Program Analysis

Before discussing program analysis with GNNs, it is important to take a step back and ask where machine learning can help program analysis and why. At a first look these two fields seem incompatible: static program analyses commonly seek guarantees (e.g., a program *never* reaches some state) and dynamic program analyses certify some aspect of a program's execution (e.g., specific inputs yield expected outputs), whereas machine learning models probabilities of events.

At the same time, the burgeoning area of machine learning for code (Allamanis et al, 2018a) has shown that machine learning can be applied to source code across a series of software engineering tasks. The premise is that although code has a deterministic, unambiguous structure, humans write code that contains patterns and ambiguous information (e.g. comments, variable names) that is valuable for understanding its functionality. It is this phenomenon that program analysis can also take advantage of.

There are two broad areas where machine learning can be used in program analysis: learning proof heuristics, and learning static or dynamic program analyses. Commonly static program analyses resort into converting the analysis task into a combinatorial search problem, such as a Boolean satisfiability problem (SAT), or

another form of theorem proving. Such problems are known to often be computationally intractable. Machine learning-based methods, such as the work of (Irving et al, 2016) and (Selsam and Bjørner, 2019) have shown the promise that heuristics can be *learned* to guide combinatorial search. Discussing this exciting area of research is out-of-scope for this . Instead, we focus on the static program analysis learning problem.

Conceptually, a specification defines a desired aspect of a program's functionality and can take many forms, from natural language descriptions to formal mathematical constructs. Traditional static program analyses commonly resort to formulating program analyses through rigorous formal methods and dynamic analyses through observations of program executions. However, defining such program analyses is a tedious, manual task that can rarely scale to a wide range of properties and programs. Although it is imperative that formal methods are used for safety-critical applications, there is a wide range of applications that miss on the opportunity to benefit from program analysis. Machine learning-based program analysis aims to address this, but sacrifice the ability to provide guarantees. Specifically, machine learning can help program analyses deal with the two common sources of ambiguities: latent specifications, and ambiguous execution contexts (e.g., due to dynamically loaded code). Program analysis learning commonly takes one of three forms, discussed next.

Specification Tuning where an expert writes a sound program analysis which may yield many false positives (false alarms). Raising a large number of false alarms leads to the analogue of Aesop's "The Boy who Cried Wolf": too many false alarms, lead to true positives getting ignored, diminishing the utility of the analysis. To address this, work such as those of (Raghothaman et al, 2018) and (Mangal et al, 2015) use machine learning methods to "tune" (or post-process) a program analysis by learning which aspects of the formal analysis can be discounted, increasing precision at the cost of recall (soundness).

Specification Inference where a machine learning model is asked to learn to predict a plausible specification from existing code. By making the (reasonable) assumption that most of the code in a codebase complies with some latent specification, machine learning models are asked to infer closed forms of those specifications. The predicted specifications can then be input to traditional program analyses that check if a program satisfies them. Examples of such models are the factor graphs of (Kremenek et al, 2007) for detecting resource leaks, the work of (Livshits et al, 2009) and (Chibotaru et al, 2019) for information flow analysis, the work of (Si et al, 2018) for generating loop invariants, and the work of (Bielik et al, 2017) for synthesizing rule-based static analyzers from examples. The type inference problem discussed in Section 22.6 is also an instance of specification inference.

Weaker specifications — commonly used in dynamic analyses — can also be inferred. For example, Ernst et al (2007) and Hellendoorn et al (2019a) aim to predict invariants (assert statements) by observing the values during execution. Tufano et al (2020) learn to generate unit tests that describe aspects of the code's behavior.

Black Box Analysis Learning where the machine learning model acts as a black box that performs the program analysis and raises warnings but never explicitly formulates a concrete specification. Such forms of program analysis have great flexibility and go beyond what many traditional program analyses can do. However, they often sacrifice explainability and provide no guarantees. Examples of such methods include DeepBugs (Pradel and Sen, 2018), Hoppity (Dinella et al, 2020), and the variable misuse problem (Allamanis et al, 2018b) discussed in Section 22.5.

In Section 22.5 and 22.6, we showcase two learned program analyses using GNNs. However, we first need to discuss how to represent programs as graphs (Section 22.3) and how to process these graphs with GNNs (Section 22.4).

22.3 A Graph Represention of Programs

Many traditional program analysis methods are formulated over graph representations of programs. Examples of such representations include syntax trees, control flow, data flow, program dependence, and call graphs each providing different views of a program. At a high level, programs can be thought as a set of heterogeneous entities that are related through various kinds of relations. This view directly maps a program to a heterogeneous directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, with each entity being represented as a node and each relationship of type r represented as an edge $(v_i, r, v_j) \in \mathcal{E}$. These graphs resemble knowledge bases with two important differences (1) nodes and edges can be deterministically extracted from source code and other program artifacts (2) there is one graph per program/code snippet.

However, deciding which entities and relations to include in a graph representation of a program is a form of feature engineering and task-dependent. Note that there is no unique or widely accepted method to convert a program into a graph representation; different representations offer trade-offs between expressing various program properties, the size of the graph representation, and the (human and computational) effort required to generate them.

In this section we illustrate one possible program graph representation inspired by (Allamanis et al, 2018b), who model each source code file as a single graph. We discuss other graph representations at the end of this section. Figure 22.1 shows the graph for a hand-crafted synthetic Python code snippet curated to illustrate a few aspects of the graph representation. A high-level explanation of the entities and relations follows; for a detailed overview of the relevant concepts, we refer the reader to programming language literature, such as the compiler textbook of (Aho et al, 2006).

Tokens A program's source code is at its most basic form a string of characters. By construction programming languages can be deterministically tokenized (lexed) into a sequence of tokens (also known as lexemes). Each token can then be represented as a node (white boxes with gray border in Figure 22.1) of "token" type. These

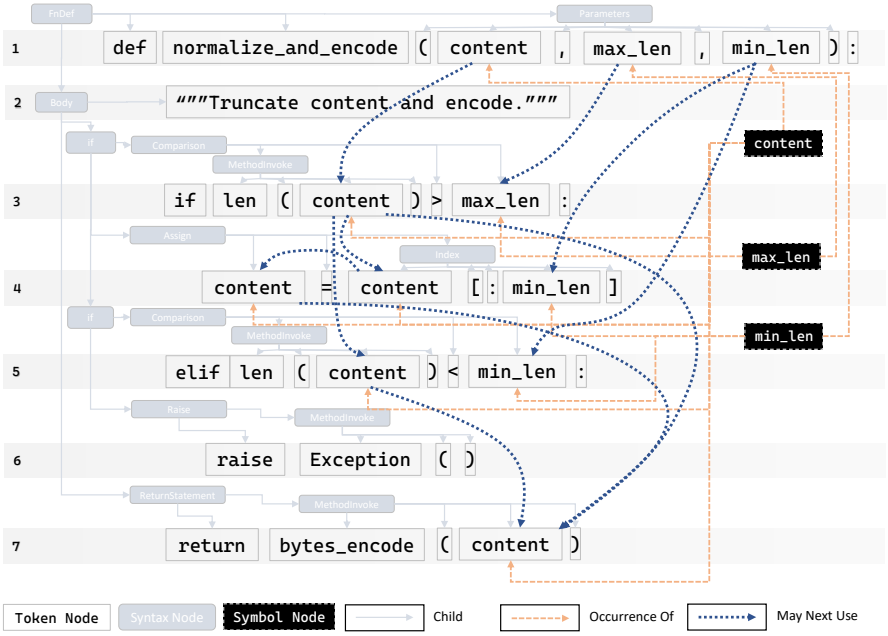

Fig. 22.1: A heterogeneous graph representation of a simple synthetic Python program (some nodes omitted for visual clarity). Source code is represented as a heterogeneous graph with typed nodes and edges (shown at the bottom of the figure). Code is originally made of tokens (token nodes) which can deterministically be parsed into a syntax tree with non-terminal nodes (vertexes). The symbols present in the snippet (e.g. variables) can then be computed (Symbol nodes) and each reference of symbol denoted by an OccurenceOf edge. Finally, dataflow edges can be computed (MayNextUse) to indicate the possible flows of values in the program. Note, the snippet here contains a bug in line 4 (see Section 22.5).

nodes are connected with a NextToken edge (not shown in Figure 22.1) to form a linear chain.

Syntax The sequence of tokens is parsed into a syntax tree. The leafs of the tree are the tokens and all other nodes of the tree are "syntax nodes" (Figure 22.1; grey blue rounded boxes). Using edges of Child type all syntax nodes and tokens are connected to form a tree structure. This stucture provides contextual information about the syntactical role of the tokens, and groups them into expressions and statements; core units in program analysis.

Symbols Next, we introduce "symbol" nodes (Figure 22.1; black boxes with dashed outline). Symbols in Python are the variables, functions, packages that are available at a given scope of a program. Like most compilers and interpreters, after parsing the code, Python creates a symbol table containing all the symbols within

each file of code. For each symbol, a node is created. Then, every identifier token (e.g., the content tokens in Figure 22.1) or expression node is connected to the symbol node it refers to. Symbol nodes act as a central point of reference among the uses of variables and are useful for modeling the long-range relationships (e.g., how an object is used).

Data Flow To convey information about the program execution we add data flow edges to the graph (dotted curved lines in Figure 22.1) using an intraprocedural dataflow analysis. Although, the actual data flow within the program during execution is unknown due to the use of branching in loops and if statements, we can add edges indicating all the valid paths that data *may* flow through the program. Take as an example the parameter min_len in Figure 22.1. If the condition in line 3 is true, then min_len will be accessed in line 4, but not in line 5. Conversely, if the condition in line 3 is false, then the program will proceed to line 5, where min_len will be accessed. We denote this information with a MayNextUse edge. This construction resembles a program dependence graph (PDG) used in compilers and conventional program analyses. In contrast to the edges previously discussed, MayNextUse has a different flavor. It does not indicate a deterministic relationship but sketches all possible data flows during execution. Such relationships are central in program analyses where existential or universal properties of programs need to be computed. For example, a program analysis may need to compute that *for all* ($\forall$) possible execution paths some property is true, or that there exists ($\exists$) at least one possible execution with some property.

It is interesting to observe that just using the token nodes and NextToken edges we can (deterministically) compute all other nodes and edges. Compilers do exactly that. Then why introduce those additional nodes and edges and not let a neural network figure them out? Extracting such graph representations is cheap computationally and can be performed using the compiler/interpreter of the programming language without substantial effort. By directly providing this information to machine learning models — such as GNNs — we avoid "spending" model capacity for learning deterministic facts and introduce inductive biases that can help on program analysis tasks.

Alternative Graph Representations So far we presented a simplified graph representation inspired from (Allamanis et al, 2020). However, this is just one possible representation among many, that emphasizes the local aspects of code, such as syntax, and intraprocedural data flow. These aspects will be useful for the tasks discussed in Sections 22.5 and 22.6. Others entities and relationships can be added, in the graph representation of Figure 22.1. For example, Allamanis et al (2018b) use a GuardedBy edge type to indicate that a statement is guarded by a condition (i.e., it is executed only when the condition is true), and Cvitkovic et al (2018) use a SubtokenOf edge to connect tokens to special subtoken nodes indicating that the nodes share a common subtoken (e.g., the tokens max_len and min_len in Figure 22.1 share the len subtoken).

Representations such as the one presented here are *local*, i.e. emphasize the local structure of the code and allow detecting and using fine-grained patterns. Other local

representations, such as the one of (Cummins et al, 2020) emphasize the data and control flow removing the rich natural language information in identifiers and comments, which is unnecessary for some compiler program analysis tasks. However, such local representations yield extremely large graphs when representing multiple files and the graphs become too large for current GNN architectures to meaningfully process (e.g., due to very long distances among nodes). Although a single, general graph representation that includes every imaginable entity and relationship would seem useful, existing GNNs would suffer to process the deluge of data. Nevertheless, alternative graph constructions that emphasize different program aspects are found in the literature and provide different trade-offs.

One such representation is the *global* hypergraph representation of (Wei et al, 2019) that emphasizes the inter- and intraprocedural type constraints among expressions in a program, ignoring information about syntactic patterns, control flow, and intraprocedural data flow. This allows processing whole programs (instead of single files; as in the representation of Figure 22.1) in a way that is suitable for predicting type annotations, but misses the opportunity to learn from syntactic and control-flow patterns. For example, it would be hard argue for using this representation for the variable misuse bug detection discussed in Section 22.5.

Another kind of graph representations is the *extrinsic* one defined by (Abdelaziz et al, 2020) who combine syntactic and semantic information of programs with metadata such as documentation and content from question and answer (Q&A) websites. Such representations often de-emphasize aspects of the code structure focusing on other natural language and social elements of software development. Such a representation would be unsuitable for the program analyses of Sections 22.5 and 22.6.

22.4 Graph Neural Networks for Program Graphs

Given the predominance of the graph representations for code, a variety of machine learning techniques has been employed for program analyses over program graphs, well before GNNs got established in the machine learning community. In these methods, we find some of the origins and motivations for GNNs.

One popular approach has been to project the graph into another simpler representation that other machine learning methods can accept as input. Such projections include sequences, trees, and paths. For example, Mir et al (2021) encode the sequences of tokens around each variable usage to predict its type (as in the usecase of Section 22.6). Sequence-based models offer great simplicity and have good computational performance but may miss the opportunity to capture complex structural patterns such as data and control flow.

Another successful representation is the extraction of paths from trees or graphs. For example, Alon et al (2019a) extract a sample of the paths between every two terminal nodes in an abstract syntax tree, which resembles random walk methods (Vishwanathan et al, 2010). Such methods can capture the syntactic informa-

tion and learn to derive some of code's semantic information. These paths are easy to extract and provide useful features to learn about code. Nevertheless, they are lossy projections of the entities and relations within a program, that a GNN can – in principle – use in full.

Finally, factor graphs, such as conditional random fields (CRF) work directly on graphs. Such models commonly include carefully constructed graphs that capture only the relevant relationships. The most prominent example in program analysis includes the work of Raychev et al (2015) that captures the type constraints among expressions and the names of identifiers. While such models accurately represent entities and relationships, they commonly require manual feature engineering and cannot easily learn "soft" patterns beyond those explicitly modeled.

Graph Neural Networks GNNs rapidly became a valuable tool for learned program analyses given their flexibility to learn from rich patterns and the easiness of combining them with other neural network components. Given a program graph representation, GNNs compute the network embeddings for each node, to be used for downstream tasks, such as those discussed in Section 22.5 and 22.6. First, each entity/node v_i is embedded into a vector representation $\mathbf{n}_{v_i}$. Program graphs have rich and diverse information in their nodes, such as meaningful identifier names (e.g. max_len). To take advantage of the information within each token and symbol node, its string representation is subtokenized (e.g. "max", "len") and each initial node representation $\mathbf{n}_{v_i}$ is computed by pooling the embeddings of the subtokens, i.e., for a node v_i and for sum pooling, the input node representation is computed as

$$\mathbf{n}_{v_i} = \sum_{s \in \textsc{SubTokenize}(v_i)} \mathbf{t}_s$$

where $\mathbf{t}_s$ is a learned embedding for a subtoken s. For syntax nodes, their initial state is the embedding of the type of the node. Then, any GNN architecture that can process directed heterogeneous graphs[1] can be used to compute the network embeddings, i.e.,

$$\{\mathbf{h}_{v_i}\} = \textsc{Gnn}\left(\mathscr{G}', \{\mathbf{n}_{v_i}\}\right), \tag{22.1}$$

where the GNN commonly has a fixed number of "layers" (e.g. 8), $\mathscr{G}' = (\mathscr{V}, \mathscr{E} \cup \mathscr{E}_{inv})$, and $\mathscr{E}_{inv}$ is the set of inverse edges of $\mathscr{E}$, i.e., $\mathscr{E}_{inv} = \left\{(v_j, r^{-1}, v_i), \forall (v_i, r, v_j) \in \mathscr{E}\right\}$. The network embeddings $\{\mathbf{h}_{v_i}\}$ are then the input to a task-specific neural network. We discuss two tasks in the next sections.

[1] GGNNs (Li et al, 2016b) have historically been a common option, but other architectures have shown improvements (Brockschmidt, 2020) over plain GGNNs for some tasks.

22.5 Case Study 1: Detecting Variable Misuse Bugs

We now focus on a black box analysis learning problem that utilizes the graph representation discussed in the previous section. Specifically, we discuss the variable misuse task, first introduced by (Allamanis et al, 2018b) but employ the formulation of (Vasic et al, 2018). A variable misuse is the incorrect use of one variable instead of another already in the scope. Figure 22.1 contains such a bug in line 4, where instead of min_len, the max_len variable needs to be used to correctly truncate the content. To tackle this task a model needs to first *localize* (locate) the bug (if one exists) and then suggest a repair.

Such bugs happen frequently, often due to careless copy-paste operations and can often be though as "typos". Karampatsis and Sutton (2020) find that more than 12% of the bugs in a large set of Java codebases are variable misuses, whereas Tarlow et al (2020) find 6% of Java build errors in the Google engineering systems are variable misuses. This is a lower bound, since the Java compiler can only detect variable misuse bugs though its type checker. The author conjectures — from his personal experience — that many more variable misuse bugs arise during code editing and are resolved before being committed to a repository.

Note that this is a black box analysis learning task. No explicit specification of what the user tries to achieve exists. Instead the GNN needs to infer this from common coding patterns, natural language information within comments (like the one in line 2; Figure 22.1) and identifier names (like min, max, and len) to reason about the presence of a likely bug. In Figure 22.1 it is reasonable to assume that the developer's intent is to truncate content to max_len when it exceeds that size (line 4). Thus, the goal of the variable misuse analysis is to (1) localize the bug (if one exists) by pointing to the buggy node (the min_len token in line 4), and (2) suggest a repair (the max_len symbol).

To achieve this, assume that a GNN has computed the network embeddings $\{\mathbf{h}_{v_i}\}$ for all nodes $v_i \in \mathcal{V}$ in the program graph $\mathcal{G}$ (Equation 22.1). Then, let $\mathcal{V}_{vu} \subset \mathcal{V}$ be the set of token nodes that refer to variable usages, such as the min_len token in line 4 (Figure 22.1). First, a localization module aims to pinpoint which variable usage (if any) is a variable misuse. This is implemented as a pointer network (Vinyals et al, 2015) over $\mathcal{V}_{vu} \cup \{\emptyset\}$ where $\emptyset$ denotes the "no bug" event with a learned $\mathbf{h}_\emptyset$ embedding. Then using a (learnable) projection $\mathbf{u}$ and a softmax, we can compute the probability distribution over $\mathcal{V}_{vu}$ and the special "no bug" event,

$$p_{loc}(v_i) = \underset{v_j \in \mathcal{V}_{vu} \cup \{\emptyset\}}{\text{softmax}} \left(\mathbf{u}^\top \mathbf{h}_{v_i} \right). \tag{22.2}$$

In the case of Figure 22.1, a GNN detecting the variable misuse bug in line 4, would assign a high p_{loc} to the node corresponding to the min_len token, which is the location of the variable misuse bug. During (supervised) training the loss is simply the cross-entropy classification loss of the probability of the ground-truth location (Equation 22.2).

```
def describe_identity_pool(self, identity_pool_id):
 identity_pool = self.identity_pools.get(identity_pool_id, None)

 if not identity_pool:
-    raise ResourceNotFoundError(identity_pool)
+    raise ResourceNotFoundError(identity_pool_id)
...
```

Fig. 22.2: A diff snippet of code with a real-life variable misuse error caught by a GNN-based model in the `https://github.com/spulec/moto` open-source project.

Repair given the location of a variable misuse bug can also be represented as a pointer network over the nodes of the symbols that are in scope at the variable misuse location v_{bug}. We define $\mathscr{V}_{s@v_{bug}}$ as the set of the symbol nodes of the alternative candidate symbols that are in scope at v_{bug}, except from the symbol node of v_{bug}. In the case of Figure 22.1 and the bug in line 4, $\mathscr{V}_{s@v_{bug}}$ would contain the content and max_len symbol nodes. We can then compute the probability of repairing the localized variable misuse bug with the symbol s_i as

$$p_{rep}(s_i) = \underset{s_j \in \mathscr{V}_{s@v_{bug}}}{\text{softmax}} \left(\mathbf{w}^\top [\mathbf{h}_{v_{bug}}, \mathbf{h}_{s_i}] \right),$$

i.e., the softmax of the concatenation of the node embeddings of v_{bug} and s_i, projected onto a $\mathbf{w}$ (i.e., a linear layer). For the example of Figure. 22.1, $p_{rep}(s_i)$ should be high for the symbol node of max_len, which is the intended repair for the variable misuse bug. Again, in supervised training, we minimize the cross-entropy loss of the probability of the ground-truth repair.

Training When a large dataset of variable misuse bugs and the relevant fixes can be mined, the GNN-based model discussed in this section can be trained in a supervised manner. However, such datasets are hard to collect at the scale that existing deep learning methods require to achieve reasonable performance. Instead work in this area has opted to automatically insert random variable misuse bugs in code scraped from open-source repositories — such as GitHub — and create a corpus of randomly inserted bugs (Vasic et al, 2018; Hellendoorn et al, 2019b). However, the random generation of buggy code needs to be carefully performed. If the randomly introduced bugs are "too obvious", the learned models will not be useful. For example, random bug generators should avoid introducing a variable misuse that causes a variable to be used before it is defined (use-before-def). Although such randomly generated corpora are not entirely representative of real-life bugs, they have been used to train models that can catch real-life bugs.

When evaluating variable misuse models — like those presented in this section — they achieve relatively high accuracy over randomly generated corpora with accuracies of up to 75% (Hellendoorn et al, 2019b). However, in the author's experi-

ence for real-life bugs — while some variable misuse bugs are recalled — precision tends to be low making them impractical for deployment. Improving upon this is an important open research problem. Nevertheless, actual bugs have been caught in practice. Figure 22.2 shows such an example caught by a GNN-based variable misuse detector. Here, the developer incorrectly passed identity_pool instead of identity_pool_id as the exception argument when identity_pool was None (no pool with the requested id could be found). The GNN-based black-box analysis seems to have learned to "understand" that it is unlikely that the developer's intention is to pass None to the ResourceNotFoundError constructor and instead suggests that it should be replaced by identity_pool_id. This is without ever formulating a formal specification or creating a symbolic program analysis rule.

22.6 Case Study 2: Predicting Types in Dynamically Typed Languages

Types are one of the most successful innovations in programming languages. Specifically, type annotations are explicit specifications over the valid values a variable can take. When a program *type checks*, we get a formal guarantee that the values of variables will *only* take the values of the annotated type. For example, if a variable has an int annotation, it must contain integers but not strings, floats, etc. Furthermore, types can help coders understand code more easily and software tools such as autocompletion and code navigation to be more precise. However, many programming languages either have to decide to forgo the guarantees provided by types or require their users to explicitly provide type annotations.

To overcome these limitations, specification inference methods can be used to predict plausible type annotations and bring back some of the advantages of typed code. This is especially useful in code with partial contexts (e.g., a standalone snippet of code in a webpage) or optionally typed languages. This section looks into Python, which provides an optional mechanism for defining type annotations. For example, content in Figure 22.1 can be annotated as content: str in line 1 to indicate that the developer expects that it will only contain string values. These annotations can then be used by type checkers, such as mypy (mypy Contributors, 2021) and other developer tools and code editors. This is the probabilistic type inference problem, first proposed by (Raychev et al, 2015). Here we use the GRAPH2CLASS GNN-based formulation of (Allamanis et al, 2020) treating this as a classification task over the symbols of the program similar to (Hellendoorn et al, 2018). Pandi et al (2020) offer an alternative formulation of the problem.

For type checking methods to operate explicit types annotations need to be provided by a user. When those are not present, type checking may not be able to function and provide any guarantees about the program. However, this misses the opportunity to probabilistically reason over the types of the program from other sources of information – such as variable names and comments. Concretely, in the example of Figure 22.1, it would be reasonable to assume that min_len and max_len

have an integer type given their names and usage. We can then use this "educated guess" to type check the program and retrieve back some guarantees about the program execution.

Such models can find multiple applications. For example, they can be used in recommendation systems that help developers annotate a code base. They may help developers find incorrect type annotations or allow editors to provide assistive features — such as autocomplete — based on the predicted types. Or they may offer "fuzzy" type checking of a program (Pandi et al, 2020).

At its simplest form, predicting types is a node classification task over the subset of symbol nodes. Let $\mathcal{V}_s$ be the set of nodes of "symbol" type in the heterogeneous graph of a program. Let also, Z be a fixed vocabulary of type annotations, along with a special Any type[2]. We can then use the node embeddings of every node $v \in \mathcal{V}_s$ to predict the possible type of each symbol.

$$p(s_j : \tau) = \underset{\tau' \in Z}{\text{softmax}} \left({E_\tau}^\top \mathbf{h}_{v_{s_j}} + b_\tau \right),$$

i.e., the inner product of each symbol node embedding with a learnable type embedding E_τ for each type $\tau \in T$ plus a learnable bias b_τ. Training can then be performed by minimizing some classification loss, such as the cross entropy loss, over a corpus of (partially) annotated code.

Type Checking The type prediction problem is a specification inference problem (Section 22.2) and the predicted type annotations can be passed to a standard type checking tool which can verify that the predictions are consistent with the source code's structure (Allamanis et al, 2020) or search for the most likely prediction that is consistent with the program's structure (Pradel et al, 2020). This approach allows to reduce false positives, but does not eliminate them. A trivial example is an identity function def foo(x): return x. A machine learning model may incorrectly deduce that x is a str and that foo returns a str. Although the type checker will consider this prediction type-correct it is hard to justify as correct in practice.

Training The type prediction model discussed in this section can be trained in a supervised fashion. By scraping large corpora of code, such as open-source code found on GitHub[3], we can collect thousands of type-annotated symbols. By stripping those type annotations from the original code and using them as a ground truth a training and validation set can be generated.

Such systems have shown to achieve a reasonably high accuracy (Allamanis et al, 2020) but with some limitations: type annotations are highly structured and sparse. For example Dict[Tuple[int, str], List[bool]] is a valid type annotation that may appear infrequently in code. New user-defined types (classes) will also appear at test time. Thus, treating type annotations as district classes of a classification problem

[2] The type Any representing the top of the type lattice and is somewhat analogous to the special UNKNOWN token used in NLP.

[3] Automatically scraped code corpora are known to suffer from a large number of duplicates (Allamanis, 2019). When collecting such corpora special care is needed to remove those duplicates to ensure that the test set is not contaminated with training examples.

```
def __init__(
  self,
- embedding_dim: float = 768,
- ffn_embedding_dim: float = 3072,
- num_attention_heads: float = 8,
+ embedding_dim: int = 768,
+ ffn_embedding_dim: int = 3072,
+ num_attention_heads: int = 8,
  dropout: float = 0.1,
  attention_dropout: float = 0.1,
```

Fig. 22.3: A diff snippet from the incorrect type annotation caught by Typilus (Allamanis et al, 2020) in the open-source fairseq library.

is prone to severe class imbalance issues and fails to capture information about the structure within types. Adding new types to the model can be solved by employing meta-learning techniques such as those used in Typilus (Allamanis et al, 2020; Mir et al, 2021), but exploiting the internal structure of types and the rich type hierarchy is still an open research problem.

Applications of type prediction models include suggesting new type annotations to previously un-annotated code but can also be used for other downstream tasks that can exploit information for a probabilistic estimate of the type of some symbol. Additionally, such models can help find incorrect type annotations provided by the users. Figure 22.3 shows such an example from Typilus (Allamanis et al, 2020). Here the neural model "understands" from the parameter names and the usage of the parameters (not shown) that the variables cannot contain floats but instead should contain integers.

22.7 Future Directions

GNNs for program analysis is an exciting interdisciplinary field of research combining ideas of symbolic AI, programming language research, and deep learning with many real-life applications. The overarching goal is to build analyses that can help software engineers build and maintain the software that permeates every aspect of our lives. Still there are many open challenges that need to be addressed to deliver upon this promise.

From a program analysis and programming language perspective a lot of work is needed to bridge the domain expertise of that community to machine learning. What kind of learned program analysis can be useful to coders? How can existing program analyses be improved using learned components? What are the inductive biases that machine learning models need to incorporate to better represent program-related concepts? How should learned program analyses be evaluated amidst the lack of large annotated corpora? Until recently, program analysis research has limited itself

to primarily using the formal structure of the program, ignoring ambiguous information in identifiers and code comments. Researching analyses that can better leverage this information may light new and fruitful directions to help coders across many application domains.

Crucially, the question of how to integrate formal aspects of program analyses into the learning process is still an open question. Most specification inference work (e.g. Section 22.6) commonly treats the formal analyses as a separate pre- or post-processing step. Integrating the two viewpoints more tightly will create better, more robust tools. For example, researching better ways to incorporate (symbolic) constraints, search, and optimization concepts within neural networks and GNNs will allow for better learned program analyses that can learn to better capture program properties.

From a software engineering research additional research is needed for the user experience (UX) of the program analysis results presented to users. Most of the existing machine learning models do not have performance characteristics that allow them to work autonomously. Instead they make probabilistic suggestions and present them to users. Creating or finding the affordances of the developer environment that allow to surface probabilistic observations and communicate the probabilistic nature of machine learning model predictions will significantly help accelerate the use of learned program analyses.

Within the research area of GNNs there are many open research questions. GNNs have shown the ability to learn to replicate some of the algorithms used in common program analysis techniques (Veličković et al, 2019) but with strong supervision. How can complex algorithms be learned with GNNs using just weak supervision? Additionally, existing techniques often lack the representational capabilities of formal methods. Combinatorial concepts found in formal methods, such as sets and lattices lack direct analogues in deep learning. Researching richer combinatorial — and possibly non-parametric — representations will provide valuable tools for learning program analyses.

Finally, common themes in deep learning also arise within this domain:

- The explainability of the decisions and warnings raised by learned program analyses is important to coders who need to understand them and either mark them as false positives or address them appropriately. This is especially important for black-box analyses.
- Traditional program analyses offer explicit guarantees about a program's behavior even within adversarial settings. Machine learning-based program analyses relax many of those guarantees towards reducing false positives or aiming to provide some value beyond the one offered by formal methods (e.g. use ambiguous information). However, this makes these analyses vulnerable to adversarial attacks (Yefet et al, 2020). Retrieving some form of adversarial robustness is still desirable for learned program analyses and is still an open research problem.
- Data efficiency is also an important problem. Most existing GNN-based program analysis methods either make use of relatively large datasets of annotated code (Section 22.6) or use unsupervised/self-supervised proxy objectives (Sec-

tion 22.5). However, many of the desired program analyses do not fit these frameworks and would require at least some form of weak supervision.
Pre-training on graphs is one promising direction that could address this problem, but has so far is focused on homogeneous graphs, such as social/citation networks and molecules. However, techniques developed for homogeneous graphs, such as the pre-training objectives used, do *not* transfer well to heterogeneous graphs like those used in program analysis.

- All machine learning models are bound to generate false positive suggestions. However when models provide well-calibrated confidence estimates, suggestions can be accurately filtered to reduce false positives and their confidence better communicated to the users. Researching neural methods that can make accurate and calibrated confidence estimates will allow for greater impact of learned program analyses.

Acknowledgements The author would like to thank Earl T. Barr for useful discussions and feedback on drafts of this chapter.

Editor's Notes: Program analysis is one of the important downstream tasks of graph generation (Chapter 11). The main challenging problem of program analysis lies in graph representation learning (Chapter 2), which integrates the relationships and entities of the program. On basis of these graph representations, heterogeneous GNN (Chapter 16) and other variants can be used to learn the embedding of each node for task-specific neural networks. It has achieved state-of-art performances in bug detection and probabilistic type inference. There are also many emerging problems in program analysis, e.g. explainability (Chapter 7) of decisions and warnings, and adversarial robustness (Chapter 8).