LEARNING TO REPRESENT PROGRAMS WITH HETEROGENEOUS GRAPHS

Anonymous authors
Paper under double-blind review

ABSTRACT

Program source code contains complex structure information, which can be represented in structured data forms like trees or graphs. To acquire the structural information in source code, most existing researches use abstract syntax trees (AST). A group of works add additional edges to ASTs to convert source code into graphs and use graph neural networks to learn representations for program graphs. Although these works provide additional control or data flow information to ASTs for downstream tasks, they neglect an important aspect of structure information in AST itself: the different types of nodes and edges. In ASTs, different nodes contain different kinds of information like variables or control flow, and the relation between a node and all its children can also be different.

To address the information of node and edge types, we bring the idea of heterogeneous graphs to learning on source code and present a new formula of building heterogeneous program graphs from ASTs with additional type information for nodes and edges. We use the ASDL grammar of programming language to define the node and edge types of program graphs. Then we use heterogeneous graph neural networks to learn on these graphs. We evaluate our approach on two tasks: code comment generation and method naming. Both tasks require reasoning on the semantics of complete code snippets. Experiment results show that our approach outperforms baseline models, including homogeneous graph-based models, showing that leveraging the type information of nodes and edges in program graphs can help in learning program semantics.

1 INTRODUCTION

Program source code contains rich structure information, like the syntax structure and control or data flow. Learning from these structures has been a hot topic in the area of deep learning on source code. In recent years, instead of applying basic sequential neural models, researchers have used more complex neural networks to capture the explicit structure of source code. Most researches use abstract syntax trees (AST) as they are easy-to-acquire for most programming languages and semantically equivalent to source code.

A problem of ASTs is that they do not explicitly reflect structural information beyond syntax dependencies, like control and data flow. A viable solution is adding different types of control and data flow edges on ASTs to generate program graphs, and apply graph neural networks (GNN) on programs to learn their representations (Allamanis et al., 2018; Fernandes et al., 2019; Allamanis et al., 2020). However, these approaches do not consider that apart from control or data flow edges, the nodes and edges of the original ASTs are also differently typed. For example, in ASTs, some nodes refer to identifiers, and some nodes define upper-level structures as control flows. For parent-child links, the relation between a function definition node to its function body or one of its arguments is apparently different. We believe if we explicitly add node and edge types to programs graphs, it will help neural models to understand programs better.

Our idea of adding types to nodes and edges in AST coincides with the concept of heterogeneous graphs. Heterogeneous graphs, or heterogeneous information networks (Shi et al., 2016), refer to a group of graphs with multiple types of nodes and edges. A typical example of heterogeneous graphs is knowledge graphs, in which the nodes are different types of entities, and the edges represent different relations. In this paper, we propose an approach for building heterogeneous program graphs

from ASTs. To obtain the type of AST nodes and edges, we use the abstract syntax description language (ASDL) (Wang et al., 1997) grammar.

After we acquire heterogeneous graphs for code snippets, we need to find a GNN model to effectively represent these graphs. Although some existing GNN-for-code works (Fernandes et al., 2019; Allamanis et al., 2020) have pointed out that there exist different types for AST nodes, they only consider node type in the initial node embedding and neglect their differences in the message passing (Gilmer et al., 2017) step. So we turn our sight to the field of heterogeneous graph embeddings. Recently, heterogeneous graph neural networks have become widely used in heterogeneous graph embedding. Unlike traditional graph neural networks, heterogeneous graph neural networks are capable of integrating node and edge type information in the message passing stage and map different types of nodes to different feature space. We use heterogeneous graph transformer (HGT) (Hu et al., 2020b) on our heterogeneous program graphs to calculate the representation of programs.

We evaluate our approach on two tasks: comment generation and method naming, with two Python datasets from different domains. These two tasks can be seen as two different forms of code summarization, so both of them require understanding the semantics of the input code snippets. The results show that our approach outperforms existing GNN models and other state-of-the-art approaches, indicating the extra benefit of bringing heterogeneous graph information to source code.

To summarize, our contributions are: (1) To our knowledge, we are the first to put forward the idea of representing programs as heterogeneous graphs and apply heterogeneous GNN on source code snippets. (2) We propose an approach of using ASDL grammars to build heterogeneous program graphs from program ASTs. (3) We evaluate our approach on two different tasks involving graph-level prediction on source code snippets. Our approach outperforms other GNN models on both comment generation and method naming tasks.

2 Related Work

Graph Neural Networks on Program Code: Allamanis et al. (2018) first proposed an approach for learning representation for programs with graph neural networks. They create program graphs by adding edges representing data flows to ASTs, and use gated graph neural networks (GGNN) (Li et al., 2016) to learn representations for program graph nodes. They evaluated their approach on two node prediction tasks: variable naming and identifying variable misuse. Similar approaches with AST-based program graphs and GGNN have been applied on multiple tasks in the following researches, including code summarization (Fernandes et al., 2019), code expression generation (Brockschmidt et al., 2019), learning code edit (Yin et al., 2019) and variable type inference (Allamanis et al., 2020). Si et al. (2018) applied a variant of graph convolutional network (GCN) (Kipf & Welling) on augmented ASTs as a memory of an encoder-decoder model to generate loop invariants for program verification. Cvitkovic et al. (2019) addresses the open vocabulary problem in source code by adding a graph-structured cache to AST and evaluated multiple GNN models on cache-augmented ASTs for fill-in-the-blank code completion and variable naming. Wang et al. (2020) used graph matching network (Li et al., 2019) to learn the similarity of program graph pairs for code clone detection. Dinella et al. (2020) used graph isomorphism network (GIN) (Xu et al., 2019) for Javascript programs repair. Wei et al. (2019) extract "type dependency graphs" from TypeScript programs and proposed a variant of graph attention network for type inference.

Heterogeneous Graph Neural Networks: Zhang et al. (2019) proposed heterogeneous graph neural network (HetGNN), which uses random walk to sample neighbours and an LSTM to aggregate features for them. Wang et al. (2019) proposed heterogeneous graph attention network (HAN), which extends graph attention networks to heterogeneous graphs with type-specific nodel-level attention and semantic-level attention based on meta-paths. Hu et al. (2020b) proposed heterogeneous graph transformer (HGT) which leverages multi-head attention based on meta relations. HGT has achieved state-of-the-art results on multiple link prediction tasks on web-scale graphs.

Deep Learning for Code Summarization: Allamanis et al. (2016) first proposed method naming as an extreme form of code summarization, and proposed a convolutional attention network to solve this task. Hu et al. (2018) generate natural language code comments with a seq2seq model from serialized ASTs. Fernandes et al. (2019) first use graph neural networks on code comment generation and method naming. Alon et al. (2019) proposed the CODE2SEQ model for Java method nam-

ing, which encodes source code by extracting paths from ASTs. Cai et al. (2020) proposed a type auxiliary guiding encoder-decoder model with a type-associated tree-LSTM encoder for code summarization and achieved state-of-the-art results on multiple SQL-to-NL and code-to-NL datasets. Ahmad et al. (2020) combines a transformer (Vaswani et al., 2017) encoder-decoder model with copying mechanism (See et al., 2017) and relative position representations (Shaw et al., 2018). They achieved state-of-the-art results on large-scale comment generation tasks in Java and Python.

3 APPROACH

In this section, we will introduce our procedure of generating heterogeneous program graphs (HPG) from source code, and how to apply heterogeneous graph neural networks on heterogeneous program graphs.

3.1 HETEROGENEOUS PROGRAM GRAPHS

We build heterogeneous program graphs from program ASTs with the help from abstract syntax description language (ASDL) grammar. Figure 1(a) demonstrates an excerpt for the Python ASDL grammar [1]. An ASDL grammar is similar to a context-free grammar (CFG), but with two more types of important information: type and field. There are two categories of types in ASDL grammars: composite type and primitive type. Each composite type defines a group of constructors (e.g. in Figure 1(a), composite type `stmt` defines constructors `FunctionDef`, `If`, ...), and a constructor specifies a group of fields. In a constructor, each field is labeled with a unique name, and also decorated by a qualifier (single, optional (?) or sequential (*)), which denotes the valid number of elements in that field. As ASDL grammars contain rich syntactic information, it has been successfully applied to code generation and semantic parsing (Rabinovich et al., 2017; Yin & Neubig, 2018).

An AST can be built by applying a sequence of ASDL constructors. Figure 1(b) shows an example of an ASDL AST in the form of a heterogeneous graph. We can see that all nodes are assigned with a type (the left half) and a value (the right half). Here each non-terminal node corresponds to a constructor, and each terminal node corresponds to a value with a primitive type. We assign node values with constructor names or terminal token values and use their composite/primitive type as node types for heterogeneous graphs. Each parent-child relationship belongs to a specific field in the constructor of the parent node, so we associate each parent-child edge with their ASDL field name. In practice (e.g., the Python AST), some nodes only have type information but do not have node name (like node `arg` in Figure 1(b). This happens when a composite type only defines a single constructor without a name), we set their node value the same as their type.

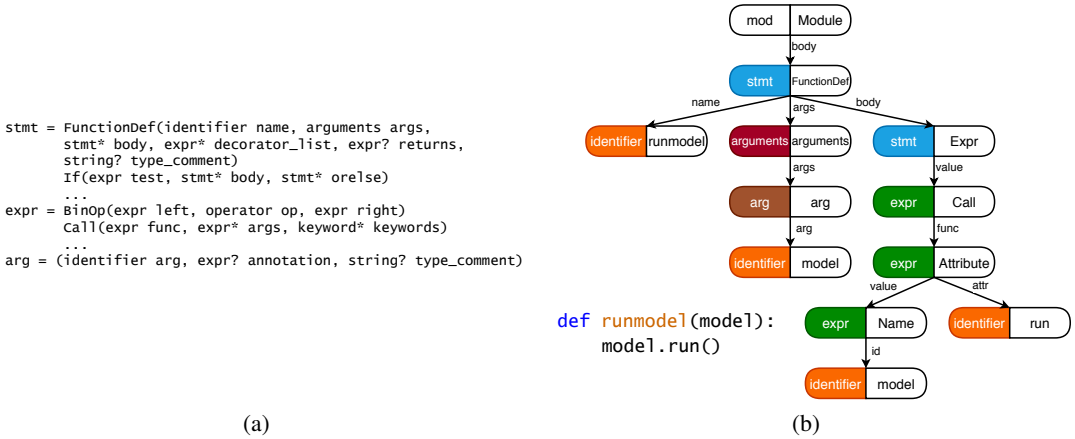

Figure 1: An example of the ASDL grammar of Python and an ASDL AST.

[1] Defined in https://docs.python.org/3/library/ast.html

We further present two simple examples to demonstrate the value of representing ASTs as heterogeneous graphs. Figure 2 (a) shows an AST subtree for an expression `a-b`. The left and right subtree of the `BinOp` node have different fields (`left` and `right`). In GNNs, these two subtree are treated equally as the neighbour of `BinOp`, which can make the model difficult to distinguish between the semantics of `a-b` from `b-a`. With typed edges and heterogeneous graph neural networks, we are able to let these two subtrees pass different messages to the `BinOp` node, making the GNN model capable of reasoning on the order of oprands. Figure 2 (b) shows that sometimes edges of the same type can be connected to different types of nodes. `If`, `For` and `Lambda` nodes all have a field named `body`, but the semantics of the `body` field vary between the change of the node it connect to. Generally, for differently typed nodes like `If` and `Lambda`, the semantic difference between their `body` field is larger than the difference between the body fields for `If` and `For`. If we want to address these subtle differences in the message passing stage of GNNs, we need to provide node type information to models along with edge types.

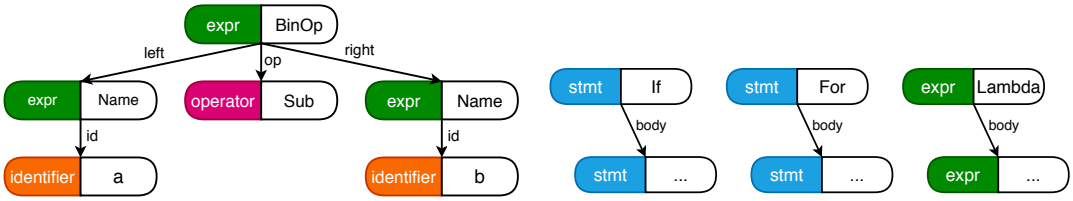

Figure 2: Two examples demonstrating the effectiveness of edge and node types in ASTs.

In addition to AST edges, we follow previous works (Allamanis et al., 2018; Brockschmidt et al., 2019) to add `NextToken` edges to the program graph. A `NextToken` edge connects a terminal node to the next terminal by the order of program text. For each edge in the heterogeneous program graph, we add a backward edge with a new edge type (e.g. the backward edge of a `body` edge is of type `body_reverse`) to improve the connectivity of graphs.

3.2 Heterogeneous Graph Transformer

We use hetereogeneous graph transformer (HGT) (Hu et al., 2020b), an attention-based heterogeneous graph neural network, to learn representation for program graphs. A heterogeneous graph $G = (\mathcal{V}, \mathcal{E}, \mathcal{A}, \mathcal{R})$ consists of a node set $\mathcal{V}$ and an edge set $\mathcal{E}$. The type of each node $\tau(n)$ and edge $\phi(e)$ in the graph belongs to the node type set $\mathcal{A}$ and edge type set $\mathcal{R}$.

An HGT layer consists of three components: heterogeneous mutual attention, heterogeneous message passing, and target-specific aggregation. The heterogeneous mutual attention is similar to the multi-head attention in transformer (Vaswani et al., 2017). For an edge $e = (s, t)$, its attention is computed by:

$$\textbf{Attention}(s, e, t) = softmax(\Big\|_{i\in[1,h]} att\_head^i(s, e, t)) \tag{1}$$

$$att\_head^i(s, e, t) = (K^i(s) W^{ATT}_{\phi(e)} Q^i(t)^T) \cdot \frac{\mu_{\langle\phi(e)\rangle}}{\sqrt{d}} \tag{2}$$

$$K^i(s) = K\_Linear^i_{\tau(s)}(H^{l-1}[s]) \tag{3}$$

$$Q^i(t) = Q\_Linear^i_{\tau(t)}(H^{l-1}[t]) \tag{4}$$

The Keys and Queries are computed based on the type of source node s or target node t. Here $H^l[s]$ is the state of node s at the l-th HGT layer. h is the number of attention heads. Then, we compute the message for e:

$$\mathbf{Message}(s,e,t) = \Big\|_{i\in[1,h]} msg\_head^i(s,e,t) \tag{5}$$

$$msg\_head^i(s,e,t) = M\_Linear^i_{\tau(s)}(H^{l-1}[s])W^{msg}_{\phi(e)} \tag{6}$$

Finally, HGT aggregate the message information with attention scores, and update node hidden states with a residual connection:

$$\widetilde{H}^{(l)}[t] = \sum_{\forall s\in N(t)} (\mathbf{Attention}(s,e,t)\cdot \mathbf{Message}(s,e,t)) \tag{7}$$

$$H^{(l)}[t] = \mathrm{A\_Linear}_{\tau(t)}(\sigma(\widetilde{H}^{(l)}[t])) + H^{(l-1)}[t] \tag{8}$$

A key difference of HGT from previous GNN models is that HGT utilizes positional encodings (Vaswani et al., 2017) to model the temporal order of nodes. In our approach, we assign a fixed timestamp $T(s)$ to each node s, which is defined by its position in the depth-first, left-to-right traversal of its AST. We adopt sinusoid functions for positional encodings and add them to the initial node embeddings as the input of the first HGT layer:

$$Base(T(s), 2i) = sin(T(s)/10000^{\frac{2i}{d}}) \tag{9}$$

$$Base(T(s), 2i+1) = cos(T(s)/10000^{\frac{2i+1}{d}}) \tag{10}$$

$$PE(T(s)) = \mathrm{T\_Linear}(T(s)) \tag{11}$$

$$H^0[s] = \mathrm{embed}(s) + PE(T(s)) \tag{12}$$

4 Experiments

4.1 Datasets and Metrics

We apply two different tasks to evaluate our program representation framework. The first one is code comment generation. For this task, we use the CoNaLa (Yin et al., 2018) dataset, which contains 2,879 Python code-NL query pairs mined from StackOverflow. CoNaLa has been evaluated by multiple previous works for code generation (Yin & Neubig, 2019; Ye et al., 2019) and comment generation (Ye et al., 2019; Cai et al., 2020).

The second task is method naming, where we predict a suitable name for a method. We choose the `ogbg-code` dataset from open graph benchmark (OGB) (Hu et al., 2020a). Each sample in `ogbg-code` consists of a method definition, and a method name split into sub-tokens. As our approach requires building heterogeneous graphs, we do not use the off-the-shelf graphs in the dataset, but create our own graphs from source code instead. We list the statistics of our datasets in Table 1. Apart from the statistics on traditional graph structures, we also show the average number of three most frequent node types in our datasets: `stmt`, `expr` and `identifier`. We can see that these two datasets are greatly different on many aspects. In CoNaLa, each code sample only contains a single line of code and do not contain complex control flows. This result in its graph scales smaller than `ogbg-code`, and the proportion of `stmt` nodes are smaller. For output tokens, the lengths of than `ogbg-code` are much shorter than CoNaLa. As we do not perform node compression for ASTs, our number of nodes in `ogbg-code` is slightly larger than reported in Hu et al. (2020a). In our experiments we use 8 node types and 114 edge type (including inverse edge types and NextToken) to build graphs for the datasets.

For both tasks, we report the results in ROUGE-L (Lin, 2004) and F1. We additionally report the BLEU-4 (Papineni et al., 2002) score for the comment generation tasks and exact matching accuracy for method naming. Notice that we follow Alon et al. (2019); Hu et al. (2020a) and calculate the F1 on bag-of-words, so different from other metrics, F1 score do not consider the order of the output tokens.

Table 1: Statistics of our experiment datasets.

	CoNaLa	ogbg-code
Train	2,279	407,976
Valid	100	22,817
Test	500	21,948
Avg. nodes in code	16.8	135.2
Avg. edges in code	41.1	364.9
Avg. tokens in summary	13.9	2.2
Avg. stmt nodes	1.1	12.7
Avg. expr nodes	8.0	60.1
Avg. identifier nodes	5.7	49.2

4.2 Implementation

We use the same encoder-decoder model for both our tasks. For the GNN encoders, we stack the GNN models for 8 layers. We follow Fernandes et al. (2019) and use a LSTM with pointer mechanism (See et al., 2017) as the decoder. The decoder calculates attention scores from the states of the input graph in the final GNN layer. As the goal of the decoder's pointer mechanism is to copy input tokens (usually identifier names) into output sequences, we do not calculate attention on all nodes, but only on terminal token nodes.

We compare our approach with several existing GNN models based on homogeneous graphs, including GGNN (Li et al., 2016) and R-GCN (Schlichtkrull et al., 2018). For all GNN models, we keep the decoder unchanged and use the same graph constructing strategy as our proposed approach. For models on homogeneous graphs, we remove the node type information but keep the edge type information since all our GNN baselines are capable of handling different edge types. As previous GNN-for-code works (Allamanis et al., 2018; Fernandes et al., 2019) did not use AST edge types, we also report result of baseline models on graphs without AST edge types (all AST edges are typed with `parent-child`). We also compare with state-of-the-art approaches for code summarization, including TAG (Cai et al., 2020), the current state-of-the-art on CoNaLa comment generation, and TransCodeSum (Ahmad et al., 2020), the current state-of-the-art on datasets from Github [2]. For the method naming task, we additionally include the results from the official OGB baselines (Hu et al., 2020a) since we used a different approach to create program graphs for this task. We implement all models in PyTorch with the graph neural network library DGL [3].

4.3 Results and Analysis

Table 2 and 3 separately shows the experiment results on comment generation and method naming. Results show that on both tasks, combining heterogeneous program graphs with HGT makes a substantial improvement over other GNN models based on homogeneous or partially-homogeneous (since they still use typed edges) graphs. Within GNN baseline models, GGNN and R-GCN achieve similar performances, with R-GCN a little worse. On CoNaLa, we achieve performances comparable to the state-of-the-art approach TransCodeSum, with a higher ROUGE-L and slightly lower BLEU and F1. On method naming, our approach outperforms all baselines and achieves the new state-of-the-art on `ogbg-code`. Unlike CoNaLa, TransCodeSum performs poorly on the `ogbg-code` dataset, which is worse than GNN baselines. On `ogbg-code`, our GNN baselines are outperformed by Hu et al. (2020a), showing that the improvement of our approach on this task comes from the heterogeneous type information and HGT, not the differences in basic graph structures. For GNN baseline models, in most experiments, their performances improve when given AST edge types based on ASDL fields. Although these models cannot handle node type information, they can still benefit from edge types for learning on source code tasks.

[2] As Cai et al. (2020) did not release their source code, we only report their results on CoNaLa as described in their paper. Ahmad et al. (2020) split source code tokens by CamelCase and snake_case during preprocessing, but we do not perform token splitting in our approach. So we reproduced their model without token splitting.

[3] https://www.dgl.ai/

Table 2: Results on the comment generation task.

	BLEU	ROUGE-L	F1
GGNN w/ AST edge type	11.8	23.0	16.7
GGNN w/o AST edge type	11.6	21.4	15.0
R-GCN w/ AST edge type	11.1	20.4	17.9
R-GCN w/o AST edge type	11.1	18.5	13.6
TAG (Cai et al., 2020)	14.1	31.8	-
TransCodeSum (Ahmad et al., 2020)	**16.4**	29.0	**30.5**
HPG+HGT (ours)	16.2	**32.1**	26.5

Table 3: Results on the method naming task.

	ROUGE-L	F1	Accuracy
GGNN w/ AST edge type	32.19	32.00	33.70
GGNN w/o AST edge type	32.07	31.90	33.70
R-GCN w/ AST edge type	26.99	27.86	29.71
R-GCN w/o AST edge type	27.90	28.54	29.77
GCN (Hu et al., 2020a)	-	32.63	-
GIN (Hu et al., 2020a)	-	32.04	-
TransCodeSum (Ahmad et al., 2020)	21.51	21.95	9.23
HPG+HGT (ours)	**34.28**	**36.15**	**38.94**

Ablatioin Study. To study the effect of different types of edges and nodes in our approach, we perform the following types of ablations for our approach:

- Remove node type (assign all nodes with the same type) and edge type information. This helps us understand the contribution of graph "heterogeneity" for source code understanding.
- Remove the backward edges for `NextToken` edges or assign the same edge type to backward edges as to forward edges. This may provide some insight into the design of program graphs from ASTs.

Table 4 shows the ablation results on CoNaLa. We can see that removing node types or edge types both result in a drop in model performance, and removing them both cause the results to drop further. This proves that leveraging node and edge types in ASTs both help GNN models to better understanding program semantics. When we remove `NextToken` backward edges, the performance drops slightly, indicating that increasing graph connectivity is more important than feeding the exact one-directional order information to program graphs. If we use the same edge type for a forward edge and its inverse, the model also performs worse. This denotes that probably assigning different types of edges with different directions makes GNNs easier to capture the tree structure in ASTs.

Table 4: Ablation study of our approach on the CoNaLa dataset.

	BLEU	ROUGE-L	METEOR	F1
Full model	16.2	32.1		26.5
-AST node type	15.3	30.0		25.1
-AST edge type	14.9	29.5		23.8
-AST node type and edge type	14.7	29.2		21.3
-NextToken backward edges	16.0	31.1		25.9
Backward edges with same type	15.8	30.9		25.4

5 CONCLUSION & FUTURE WORK

In this paper, we put forward the idea of heterogeneity in program ASTs, and presented a framework of representing source code as heterogeneous program graphs (HPG) using ASDL grammars. By applying heterogeneous graph transformer on our HPG, our approach significantly outperforms previous GNN models on two graph-level prediction tasks for source code: comment generation and method naming.

In the future, we plan to evaluate our approach on more tasks, especially node or link prediction tasks. We would also extend our approach to other programming languages and propose new models more suited for heterogeneous program graphs.

REFERENCES

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. A transformer-based approach for source code summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 4998–5007, Online, July 2020. Association for Computational Linguistics.

Miltiadis Allamanis, Hao Peng, and Charles Sutton. A convolutional attention network for extreme summarization of source code. In *International conference on machine learning*, pp. 2091–2100, 2016.

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

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*, pp. 91–105, 2020.

Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. In *International Conference on Learning Representations*, 2019.

Marc Brockschmidt, Miltiadis Allamanis, Alexander L Gaunt, and Oleksandr Polozov. Generative code modeling with graphs. In *International Conference on Learning Representations*, 2019.

Ruichu Cai, Zhihao Liang, Boyan Xu, Zijian Li, Yuexing Hao, and Yao Chen. TAG : Type auxiliary guiding for code comment generation. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, pp. 291–301. Association for Computational Linguistics, 2020.

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

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*, 2020.

Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. Structured neural summarization. In *International Conference on Learning Representations*, 2019.

Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *International Conference on Machine Learning*, pp. 1263–1272, 2017.

Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs. *arXiv preprint arXiv:2005.00687*, 2020a.

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

Ziniu Hu, Yuxiao Dong, Kuansan Wang, and Yizhou Sun. Heterogeneous graph transformer. In *Proceedings of The Web Conference 2020*, pp. 2704–2710, 2020b.

Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*.

Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. Gated graph sequence neural networks. In Yoshua Bengio and Yann LeCun (eds.), *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

Yujia Li, Chenjie Gu, Thomas Dullien, Oriol Vinyals, and Pushmeet Kohli. Graph matching networks for learning the similarity of graph structured objects. In *International Conference on Machine Learning*, pp. 3835–3845, 2019.

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

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

Maxim Rabinovich, Mitchell Stern, and Dan Klein. Abstract syntax networks for code generation and semantic parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1139–1149, 2017.

Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne Van Den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks. In *European Semantic Web Conference*, pp. 593–607. Springer, 2018.

Abigail See, Peter J Liu, and Christopher D Manning. Get to the point: Summarization with pointer-generator networks. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1073–1083, 2017.

Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. Self-attention with relative position representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pp. 464–468, 2018.

Chuan Shi, Yitong Li, Jiawei Zhang, Yizhou Sun, and S Yu Philip. A survey of heterogeneous information network analysis. *IEEE Transactions on Knowledge and Data Engineering*, 29(1): 17–37, 2016.

Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. Learning loop invariants for program verification. In *Advances in Neural Information Processing Systems*, pp. 7751–7762, 2018.

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*, pp. 5998–6008, 2017.

Daniel C Wang, Andrew W Appel, Jeffrey L Korn, and Christopher S Serra. The zephyr abstract syntax description language. In *DSL*, volume 97, pp. 17–17, 1997.

Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 261–271. IEEE, 2020.

Xiao Wang, Houye Ji, Chuan Shi, Bai Wang, Yanfang Ye, Peng Cui, and Philip S Yu. Heterogeneous graph attention network. In *The World Wide Web Conference*, pp. 2022–2032, 2019.

Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. Lambdanet: Probabilistic type inference using graph neural networks. In *International Conference on Learning Representations*, 2019.

Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *International Conference on Learning Representations*, 2019.

Hai Ye, Wenjie Li, and Lu Wang. Jointly learning semantic parser and natural language generator via dual information maximization. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pp. 2090–2101, 2019.

Pengcheng Yin and Graham Neubig. Tranx: A transition-based neural abstract syntax parser for semantic parsing and code generation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 7–12, 2018.

Pengcheng Yin and Graham Neubig. Reranking for neural semantic parsing. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pp. 4553–4559, 2019.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. Learning to mine aligned code and natural language pairs from stack overflow. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pp. 476–486. IEEE, 2018.

Pengcheng Yin, Graham Neubig, Miltiadis Allamanis, Marc Brockschmidt, and Alexander L Gaunt. Learning to represent edits. In *International Conference on Learning Representations*, 2019.

Chuxu Zhang, Dongjin Song, Chao Huang, Ananthram Swami, and Nitesh V Chawla. Heterogeneous graph neural network. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 793–803, 2019.