Learning to Represent Programs with Heterogeneous Graphs

Kechi Zhang[*][†]
zhangkechi@pku.edu.cn
Peking University
China

Wenhan Wang[*]
wwhjacob@hotmail.com
Nanyang Technological University
Singapore

Huangzhao Zhang[†]
zhang_hz@pku.edu.cn
Peking University
China

Ge Li[†][‡]
lige@pku.edu.cn
Peking University
China

Zhi Jin[†][‡]
zhijin@pku.edu.cn
Peking University
China

ABSTRACT

Code representation, which transforms programs into vectors with semantics, is essential for source code processing. We have witnessed the effectiveness of incorporating structural information (*i.e.*, graph) into code representations in recent years. Specifically, the abstract syntax tree (AST) and the AST-augmented graph of the program contain much structural and semantic information, and most existing studies apply them for code representation. The graph adopted by existing approaches is homogeneous, *i.e.*, it discards the type information of the edges and the nodes lying within AST. That may cause plausible obstruction to the representation model. In this paper, we propose to leverage the type information in the graph for code representation. To be specific, we propose the heterogeneous program graph (HPG), which provides the types of the nodes and the edges explicitly. Furthermore, we employ the heterogeneous graph transformer (HGT) architecture to generate representations based on HPG, considering the type of information during processing. With the additional types in HPG, our approach can capture complex structural information, produce accurate and delicate representations, and finally perform well on certain tasks. Our in-depth evaluations upon four classic datasets for two typical tasks (*i.e.*, method name prediction and code classification) demonstrate that the heterogeneous types in HPG benefit the representation models. Our proposed HPG+HGT also outperforms the SOTA baselines on the subject tasks and datasets.

KEYWORDS

graph neural networks, heterogeneous graphs, code representation

*The two authors share equal contribution.
†Also with Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, China.
‡Corresponding author.

1 INTRODUCTION

Generating representations for programs is an essential process in source code processing. This process transforms programs of different formats (token sequences, abstract syntax trees, or dependency graphs, *etc.*) into vectorized or tensorized representations. Code representation learning is also the basis of many other tasks, including code summarization [19], method name prediction [6, 7], code classification [30] and type inference [3], *etc.* With powerful deep learning (DL) techniques, representations of code can be obtained via a variety of deep neural networks. In order to obtain better code representations, researchers have incorporated structural information into the modeling process – from the trivial token sequences [11, 18], to the abstract syntax trees (AST) [6, 7, 30, 56] and finally to the graphs [3, 4, 8, 14, 54]. Allamanis et al. [4] first propose to leverage graph neural networks (GNN) for learning representation of the programs. Specifically, they create program graphs based on ASTs and use the GGNN model [26] to learn representations of program graph nodes. Up to the present, we have witnessed the capability of graph-based techniques, as they could achieve the state-of-the-art (SOTA) performance in bug detection and fixing [12], clone detection [46], variable misuse prediction [48], *etc.*

Although the existing graph-based techniques have proven their usefulness, there remains one issue that can not be ignored – the existing approaches mostly neglect the semantic type information and adopt the (partially) homogeneous graph, *i.e.*, all nodes are of the same type and the edges are only categorized in a coarse-grained manner. To be clear, previous studies regard the edges in the AST as the same type and augment the AST with other kinds of edges, such as data flow edges [4], to retrieve the acquired graph. The homogeneous graph lacks sufficient type information, and it may make the model incapable of jointly learning the semantics of the nodes and the edges in the graph. The issue of lacking type information then leads to two major shortcomings in the representation models. ❶ The model cannot explicitly recognize the different type of each node. For instance, in the homogeneous graph in Figure 1(b), the model cannot distinguish the type of a, −, and b, although a along with b are identifiers and − is an operator. Ignoring the

(a) AST. The framed part is transformed into (b) and (c)

(b) Homogeneous graph　　　(c) Heterogeneous graph
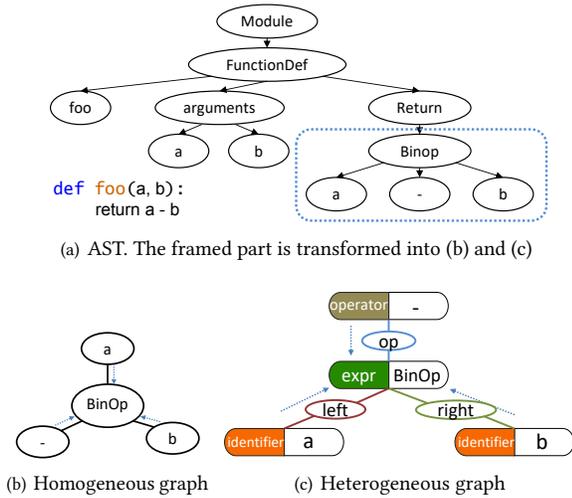
Figure 1: An illustrative example of homogeneous and heterogeneous graphs. Previous studies utilize the homogeneous graph as shown in (b) to represent the framed part in (a), while our proposed HPG provides type information of the nodes and the edges as in (c).

node types may hinder the model from sufficiently capturing the semantics of the program. ❷ Topologically, the model regards all connections of nodes (*i.e.*, the edges) to be identical, leaving much structural information discarded. For the same concrete example, when the edge type is not provided, the expressions of "a-b" and "b-a" may lead to the same homogeneous graph in Figure 1(b). In other words, the semantic meaning of a homogeneous graph can be ambiguous (both "a-b" and "b-a" are suitable for Figure 1(b)). Both of these two drawbacks are caused by missing information in the homogeneous graph, and it is probable that they obstruct the learning process of the model to a certain degree.

To tackle the issues of the homogeneous graph, in this paper, we propose to leverage heterogeneous graphs for code representation. As the name suggests, the heterogeneous graph is made up of nodes and edges of different types, *i.e.*, there are multiple types of the nodes or the edges, or both. The type information about the nodes and the edges brings forth two advantages, which may settle the aforementioned issues lying within the homogeneous graph. ❶ The types (both node and edge) are explicitly provided in the heterogeneous graph. Heuristically, the semantic information of the nodes and the edges is supposed to be clustered by their types. *E.g.*, given Figure 1(c), the representation model is informed that a and b are identifier nodes and − is an operator node. Therefore, the model can process these nodes in different manners according to their types. ❷ The edge type guarantees that the graph is not ambiguous. Also take Figure 1(c) for instance, the two edges associated with a and b are of type `left` and `right` respectively. Hence, Figure 1(c) can only refer to the expression of "a-b" rather than "b-a". This may cause the learning procedure of the model much easier.

In this paper, we propose the heterogeneous program graph (HPG) for code representation. An HPG of a certain code snippet is generated based on its AST according to the abstract syntax description language (ASDL) [44], as an example is shown in Figure 1(c).

The types in HPG allow the representation model to accurately capture the semantics and the relations of the nodes and the edges. We will explain the technical details of how to produce HPG according to the ASDL grammar in section 4.

To make full use of HPG, we employ the heterogeneous GNN as the representation model, which integrates the type information during processing the graph. In this paper, we adopt the heterogeneous graph transformer (HGT) architecture [20] to generate representations according to our proposed HPG. HGT applies the heterogeneous attention to model the edges between the nodes. Especially, during the computation of the attention, HGT uses different weights for different types of the nodes and the edges. Therefore, in short words, HGT is capable to process the nodes and the edges of different types in different manners and to capture the more accurate semantic information.

To demonstrate the usefulness of our proposed HPG and HGT (HPG+HGT), we conduct in-depth evaluations upon four datasets for two tasks (*i.e.*, method name prediction [5, 21] and code classification [33]) against currently SOTA baselines, including Code Transformer[58], Cognac [45] and other tree-structured or graph-structured models. The experimental results demonstrate that ❶ the heterogeneous types in HPG are capable to improve the performance of the GNN models, ❷ HPG+HGT outperforms current SOTA approaches in most cases, and ❸ the design of each component in HPG+HGT is necessary and may benefit the performance.

The main contributions of this paper are summarized as follow.

- To the best of our knowledge, we are the very first to uncover the type missing issue within the homogeneous graph for code representation.
- We are the very first to propose leveraging the heterogeneous program graph for code representation. Our proposed HPG is capable to relieve the type missing issue in the homogeneous graph. Specifically, HPG explicitly provides the type information of the nodes and the edges in AST.
- We propose a heterogeneous-graph-based framework based on the HGT architecture to process HPG. Our in-depth evaluations on four datasets for two classic code representation tasks certify the effectiveness and the usefulness of our proposed HPG+HGT.

2 RELATED WORK

2.1 Code Processing by Deep Learning

Deep learning has been proved useful in plenty of code processing tasks, which can be roughly divided into two categories, *i.e.*, generation tasks and classification tasks. Generation tasks take program of different formats (*e.g.*, token sequences, ASTs, data flow graphs, *etc.*) as input and produce a sequence of information, such as natural language documentations, *i.e.*, code summarization [1, 10, 14, 19, 22]) and method names, *i.e.*, method name prediction [5–7, 23, 27, 28, 31, 45, 58], *etc.* Allamanis et al. [5] first propose the convolutional attention networks to predict method names given the body. Hu et al. [19] propose DeepCom to generate comments for methods or functions. Alon et al. [6] propose code2seq for method name prediction. Zügner et al. [58] propose Code Transformer to incorporate multiple relations to learning both the structure and the context jointly, improving the performance for multiple generation tasks. As for classification tasks, the model usually transform

the program of different formats into vectorized representations and classify according to the semantics. Typical classification tasks include code classification [9, 30] and code clone detection [40, 46], *etc*. Mou et al. [30] first propose tree-based TBCNN for code functionality classification. Later, Zhang et al. [56] propose ASTNN for code classification and clone detection. Recently, Hellendoorn et al. [17] propose GREAT based on the transformer architecture with relational information from code graphs. GREAT has achieved SOTA performance on many classification tasks.

In this paper, to generally demonstrate the capacity of our proposed HPG+HGT, we carry out extensive experiments on both generation and classification tasks, *i.e.*, method name prediction [6, 58] and code classification [33].

2.2 GNN for Code Representation

The graph neural networks (GNN) [16] is a specialized deep learning model designed to model the graph-structured data, and it is widely employed on many general purposed tasks, such as molecule prediction [13, 15] and social network analysis [32, 34], *etc*. Kipf and Welling [24] propose GCN by averaging the neighbor of each node to gather information. Li et al. [26] propose GGNN, introducing the gate mechanism from GRU [11].

As for the field of code representation, the model need to process the program of different formats and produce a vectorized representation for the certain code processing tasks, as introduced in the last section. As aforementioned, the program often contain extensive structural information. The GNN architecture can naturally extract such structural information from the graph, and hence, a lot of GNN solutions to code representation emerges in the recent years [3, 4, 8, 14, 38, 46, 48, 54]. Among them, Allamanis et al. [4] first employ GGNN to process program graphs, which is augmented from AST with additional manually crafted edges. Later, the similar approaches, in which the AST-augmented graph and the GGNN architecture are adopted, are employed to tackle many code processing tasks, including code summarization [14], expression generation [8], code edition [54] and type inference [3]. However, the aforementioned approaches ignore the types of the nodes and the edges, resulting in possible ambiguous semantics of the graph representation (as illustrated in section 1).

In this paper, to tackle the issue, we introduce the heterogeneous graph with type information of the nodes and the edges for the program graph, which has been demonstrated effective for more general purposed tasks [20, 47, 55]. Specifically, Hu et al. [20] propose the heterogeneous graph transformer (HGT) architecture, utilizing multi-head attention based on meta relations, and it has already achieved SOTA on many web-scale graph tasks. Inspired by the existing heterogeneous GNN [20, 47, 55], we employ the ASDL grammar to retrieve the types of the nodes and the edges in AST, and build the heterogeneous program graph based on the AST and the additional types. In this paper, we employ HGT as the backbone of our proposed model.

3 PRELIMINARY

To facilitate understanding of HPG and HGT, we present some necessary background knowledge in this section, including the formulation of GNN, heterogeneous graph, and ASDL.

3.1 Graph Neural Networks

The GNN models have shown promising results for modeling the structural information for program codes [36, 50]. In this section, we briefly introduce the neural message passing framework [15, 51] of the GNN architecture. In general, message passing refers to neighborhood aggregation, *i.e.*, each node aggregates features of its neighbors to obtain the new node feature. Please refer to the formulation as below:

$$a_v^{(k)} = \text{Aggregate}^{(k)} \left(\{ h_u^{(k-1)} : u \in \mathcal{N}(v) \} \right) \quad (1)$$

$$h_v^{(k)} = \text{Combine}^{(k)} \left(h_v^{(k-1)}, a_v^{(k)} \right) \quad (2)$$

where the superscripts (k) or $(k-1)$ refer to the layers, the subscripts v or u refer to the nodes, $a_v^{(k)}$ and $h_v^{(k)}$ are the aggregated vector and the feature vector of the node v from the k-th layer, and $\mathcal{N}(v)$ refers to the neighbors of v. To be specific, the k-th layer aggregates (collects) feature of v's neighbors from the $k-1$-th layer, as Eq. 1 suggests, forming the aggregated vector $a_v^{(k)}$ of v in the k-th layer. Then $a_v^{(k)}$ and $h_v^{(k-1)}$ are combined, resulting in the new feature $h_v^{(k)}$ of v, as shown in Eq. 2.

Most previous GNN for code representation researches apply GNNs on AST-based or AST-augmented graphs [3, 4, 8, 12, 14, 46, 54], while a few studies explore the non-AST-based graph of code, which are constructed by type dependency [49], code property [57] for specific downstream tasks, *etc*.

3.2 Heterogeneous Graph

Heterogeneous graph. A heterogeneous graph [39] is a graph consisting of different types of entities, *i.e.*, nodes, and different types of relations, *i.e.*, edges. It is defined as a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{A}, \mathcal{R})$ where each node $v \in \mathcal{V}$ and each edge $e \in \mathcal{E}$ are associated with their type mapping functions $\tau(v) : \mathcal{V} \to \mathcal{A}$ and $\phi(e) : \mathcal{E} \to \mathcal{R}$. $\mathcal{A}$ and $\mathcal{R}$ denote the sets of the node type and the edge type, respectively. In addition, at least one of the nodes or the edges contains multiple types, *i.e.* $|\mathcal{A}| + |\mathcal{R}| > 2$. Compared with the traditional homogeneous graph, the heterogeneous graph provides additional type information of the nodes and the edges, as Figure 1 demonstrates.

Meta relation. To facilitate understanding of heterogeneous GNN, we present a brief introduction of meta relation [20]. Meta relation jointly considers the types of the edge along with its source and target nodes. To be specific, for an edge $e = (s, t)$ connecting the nodes s and t, its meta relation is defined as $\langle \tau(s), \phi(e), \tau(t) \rangle$, where τ and ϕ identify the types of s along with t, and e, respectively. Meta relation describes the pattern of the connections in the heterogeneous graph, which may help to build more accurate and delicate representations than the node or edge types alone.

3.3 Abstract Syntax Description Language

The ASDL grammar contains rich syntactic and semantic information, which has been successfully applied to code generation and semantic parsing [35, 53]. ASDL [44], which consists of a sequence of productions, is quite similar to context-free grammar (CFG). However, ASDL contains additional information, *i.e.*, the node type

```
stmt = FunctionDef(identifier name, arguments args,
         stmt* body, expr* decorator_list, expr? returns,
         string? type_comment)
       If(expr test, stmt* body, stmt* orelse)
       ...
expr = BinOp(expr left, operator op, expr right)
       Call(expr func, expr* args, keyword* keywords)
       ...
arg = (identifier arg, expr? annotation, string? type_comment)
```

(a) An example of ASDL for Python.
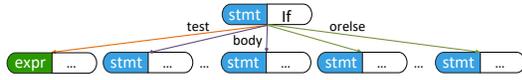
(b) An illustration of a subtree constructed by ASDL.

Figure 2: An illustrative example of ASDL and the syntax tree. The subtree is constructed according to the ASDL rule of If.

and the field, which differs it from CFG. The additional information in ASDL can be essential for us to construct heterogeneous graphs for programs. Therefore, we provide a brief introduction to ASDL in this section.

Constructor. A constructor defines a production rule in ASDL. In general, it defines the parent, the children, and their connections. Take the If constructor in the blue dashed frame in Figure 2(a) for instance, through it, a subtree in Figure 2(b) can be produced.

Node type. As the name suggests, the node type in ASDL is capable to provide the type information of the nodes for the heterogeneous graph. The node type in ASDL can be divided into two categories – composite types and primitive types. ❶ The composite type defines a group of constructors, which specifies how to construct such nodes with the certain type. *E.g.*, in Figure 2(a), constructors FunctionDef and If, *etc.*, are defined by composite type stmt, *i.e.*, the types of these nodes are stmt. ❷ The primitive type, such as identifier, int, and string, *etc.*, defines a set of terminals.

Field. The field in ASDL can be viewed as the edge types in the heterogeneous graph. The field is one of the components in the constructor, and it specifies the relation between the parent and each of its children. Take the If constructor in Figure 2(a) for instance, the connection between If and expr is of the field test, and the connection between If and the first stmt is of the field body. According to the field in ASDL, it is handy to mark the edge types in the AST to produce a heterogeneous graph.

Qualifier. The qualifier in a constructor denotes the number of children in the certain field. There are three valid qualifiers – single (one and only one), optional (?, zero or one) and sequential (*, any number). Still take If in Figure 2(a) for example, it has one expr of the field test, multiple stmt's of the field body, and multiple stmt's of the field orelse (see Figure 2(b)).

4 HETEROGENEOUS PROGRAM GRAPH

4.1 Overview of HPG

The HPG is a format of heterogeneous graph for programs generated from the AST. Formally, the HPG $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{A}, \mathcal{R}, \mathcal{X})$ consists of a node set $\mathcal{V}$, an edge set $\mathcal{E}$, a node type set $\mathcal{A}$, an edge type set $\mathcal{R}$, and a node value set $\mathcal{X}$. The node $v \in \mathcal{V}$ corresponds to an AST node, which is generated following ASDL. The edges $e_1, e_2, \cdots \in \mathcal{E}$
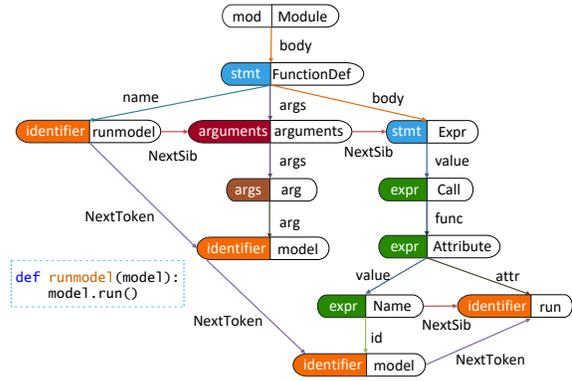
Figure 3: An example of HPG for a Python code snippet. Each node is assigned with a type (left part) and a value (right part). Each edge has a type label. We omit the reverse edges for viewing convenience.

consists of edges in the ASDL AST and rule-based manually crafted edges. The types of the nodes ($\mathcal{A}$) and the edges ($\mathcal{R}$) are similarly defined as in section 3.2. In addition, for each node $v \in \mathcal{V}$, besides its type $\tau(v) \in \mathcal{A}$, there is also a value $X(v) \in \mathcal{X}$.

Figure 3 is an illustrative example of HPG. Each node has a type (left part) and a value (right part), and each edge has a type. Specifically, the NextSib and NextToken edges are manually crafted edges in addition to the AST edges.

4.2 Generation of HPG

With the help of ASDL, we are able to build HPGs from ASTs easily. In general, there are two major steps to generate an HPG – building a typed AST and inserting manually crafted edges.

Typed AST. To build an AST with its nodes and edges typed, we have to assign the type information according to the ASDL grammar. ❶ For the node, the value is assigned with the corresponding constructor name or the terminal token value, and the type is assigned with the corresponding composite or primitive type (composite for non-terminals and primitive for terminals). In addition, for some programming languages (*e.g.*, Python), some nodes only have the type but do not possess a name (*e.g.*, the arg node in Figure 3) [1]. We set the values of this kind of nodes with their types, *i.e.*, their values and types are identical. ❷ As for the edge, as aforementioned in section 3.3, the field reflects the relation between the parent and the child, *i.e.*, the edge type. Therefore, we associate each AST edge with their corresponding ASDL field name to build a typed AST.

Manually crafted edge. In order to retain as much structural information as possible, we also insert heuristic-based manually crafted edges to build HPG, including NextSib, NextToken, and the reversed edges. ❶ NextSib points from a node to its next (right) sibling, and NextToken points from a terminal node to its next (right) terminal by the order of the text of the code snippet. We include NextSib and NextToken in HPG because previous work has already demonstrated that these additional edges are quite beneficial to the GNN model [4, 8]. Our experimental results also confirm the necessity of NextSib and NextToken. ❷ The reversed edge e_{rev} of

---

[1]This phenomenon occurs when a composite type only defines one single constructor. Under such circumstances, the ASDL grammar no longer needs to assign a name to distinguish this certain constructor.
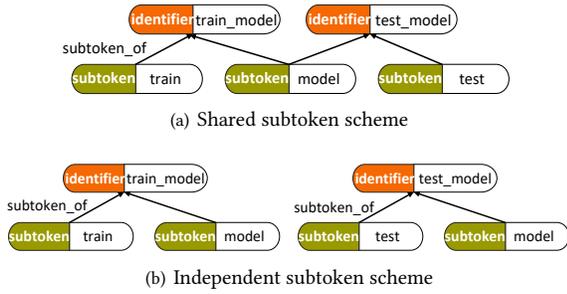
(a) Shared subtoken scheme

(b) Independent subtoken scheme

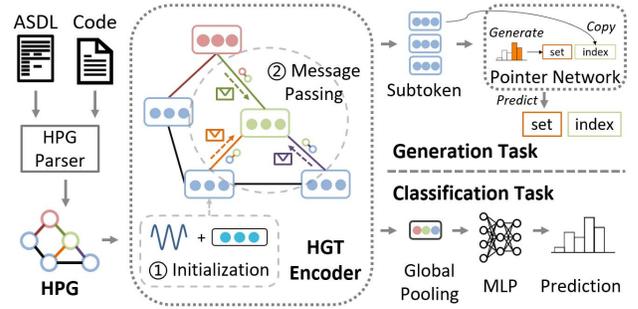Figure 4: Shared and independent schemes for subtoken.

Figure 5: Overview of the proposed HPG+HGT framework (best view in color). The HPG parser generates HPG from the code snippet according to the ASDL grammar, where the color refers to the types of the nodes and the edges. The HGT encoder takes two steps to process HPG and produce the representations – feature initialization and iterative message passing (section 5.1). At last, the representations are fed into downstream modules for certain tasks (section 5.3).

edge $e = (s, t)$, from node s to node t, reversely points from t to s, i.e., $e_{rev} = (t, s)$. In the HPG, we insert a reverse edge for every forward edge (including the AST edge, NextSib, and NextToken) to improve the connectivity of the graph. The type of the reversed edge is determined by its corresponding forward edge, e.g., the reversed edge of a body edge would be of type body_reverse and the reversed edge of NextToken would be LastToken. Heuristically, the better the connectivity, the more efficiently and effectively the GNN model can capture the useful features [2].

4.3 Integrating Subtoken Information

The identifiers, including the method names and the variable names, etc., are quite essential for code representation because they carry much information in the text as the previous studies suggest [6, 31, 45]. The identifier is often composed of several words, increasing the difficulty of extracting useful information. On the other hand, due to the identifiers, the vocabulary size of a program corpus is often extremely large. Therefore, to capture the semantics and to avoid OOV caused by large vocabulary size, we employ the subtoken technique in HPG, by splitting the terminal nodes into multiple parts based on the camel case or the snake case [2].

Subtoken. We introduce a new node type, subtoken, and a new edge type, subtoken_of, into HPG. The original terminal nodes are splitted into multiple subtoken nodes by the value, and new subtoken_of edges are inserted to indicate the origin of the subtoken nodes. Take Figure 4 for illustration, the identifier node with the value "train_model" is splitted into two subtoken nodes – "train" and "model", with two subtoken_of edges inserted from the new subtoken terminal nodes to the original "train_model" node. In addition, the reversed edge of subtoken_of is also inserted into HPG during splitting. Since there may be multiple tokens to be split, we can either share the subtoken nodes among the original identifier nodes, or separate them independently. We will discuss these two schemes in the rest of this section.

Shared subtoken. In this scheme, the identical subtoken node is shared among different identifier nodes. As demonstrated in Figure 4(a), the "model" subtoken is shared by two identifier nodes, i.e., "train_model" and "test_model". The shared scheme may

effectively reduce the size of the graph, and previous work [3] has demonstrated that similar strategy is possible to get good semantic representations on the subtoken nodes.

Independent subtoken. Another scheme is to treat the subtoken nodes of each identifier independently. Take Figure 4(b) for illustration, the subtoken nodes of "train_model" and "test_model" are independently separated. The independent scheme may obtain polysemous representations for the same subtoken, but it may suffer from large graph size as there are too many subtoken nodes.

5 HETEROGENEOUS GRAPH TRANSFORMER

We employ the HGT model, an attention-based heterogeneous graph neural network, as the encoder to generate representations upon the HPG. HGT encodes HPG to generate a vectorized representation for each node and then performs global pooling to produce the overall representation of the whole HPG. The representation of the code can be fed into downstream modules for certain tasks. For generation tasks, such as method name prediction [5–7], we leverage the pointer network [43] for sequence generation. As for classification tasks, such as code classification [30, 33], we utilize an MLP upon the overall HPG representation as the classifier. Figure 5 presents the overall workflow of our model for different tasks, and we will elaborate on each component in detail in this section.

5.1 HGT Encoder

The HGT encoder encodes HPG with the heterogeneous graph transformer architecture [20]. Specifically, the HGT encoder composes of a positional encoding layer, an embedding layer and multiple HGT layers. Similar to traditional transformer [42], an HGT layer employs heterogeneous mutual attention during message passing, in order to aggregate the information of the previous layer from the neighbors.

Positional encoding. The technique of positional encoding, which assures the temporal order of the nodes, is essential for transformer-based HGT. As the program is executed in a fixed order (e.g., different stmt nodes generally have sequential relationships that can not be easily exchanged), modeling order information in HPGs

---

[2]During the message passing process, GNN can only consider the information of the k-hop reachable neighbors of each node. The one-way forward edge only makes some nodes in the graph unreachable, e.g., no node can reach the AST root in most cases. With the help of the reversed edge, every node is allowed to reach all other nodes. Therefore, during message passing, the k-hop neighbor size becomes larger, and the GNN may capture more structural information.

is important for learning their representations. We assign a fixed timestamp to each node $v \in \mathcal{V}$, which is the position of v in the depth-first traversal sequence of the AST. After that, a sine function is conducted upon the timestamps to obtain the positional encoding of each node, which is similar to the previous work [37, 42].

Feature initialization. It is necessary to have an initialization of the node feature, *i.e.*, $h_v^{(0)}$, to initiate the message passing process. In HGT, we add the embedding of the value and the positional encoding of each node v to obtain $h_v^{(0)}$.

Heterogeneous message. The heterogeneous message gathers information from the neighbors for each node, considering the types of both the nodes and the edges. Specifically, for a node t, we would like to collect information from its neighbor $s \in \mathcal{N}(t)$ ($\mathcal{N}(t)$ is the neighbor node set of t), *i.e.*, there exists an edge $e = (s, t)$ pointing from s to t. The process is formulated as:

$$M^{(k)}(s, e, t) = \text{M-Linear}_{\tau(s)} \left(h_s^{(k-1)} \right) \cdot W_{\phi(e)}^M \quad (3)$$

where the information of s ($h_s^{(k-1)}$) is projected into the message space with M-Linear$_{\tau(s)}$, taking the node type ($\tau(s)$) into account, and then it incorporates the edge type ($\phi(e)$) dependency by $W_{\phi(e)}^M$. Each M is a message head, and HGT concatenates multiple independent heads forming the heterogeneous message, *i.e.*, Message = Concat($M_1, \cdots, M_h$) ((k) and (s, e, t) are omitted).

Heterogeneous mutual attention. The attention determines how important the heterogeneous message is during the aggregation in message passing. Similar to the previous self-attention [42], heterogeneous mutual attention computes the attention (importance) score upon the adjacent nodes in HPG, considering the types of the ndoes and the types. To be specific, for $e = (s, t)$, the unnormalized attention score is formulated as:

$$A^{(k)}(s, e, t) = \left(K^{(k)}(s) \cdot W_{\phi(e)}^A \cdot \left(Q^{(k)}(t) \right)^T \right) \cdot \frac{\mu_{\langle \tau(s), \phi(e), \tau(t) \rangle}}{\sqrt{d}} \quad (4)$$

where $Q^{(k)}(t)$ and $K^{(k)}(s)$ are the query and the key to compute attention, formulated as $Q^{(k)}(t) = \text{Q-Linear}_{\tau(t)}(h_t^{(k-1)})$ and $K^{(k)}(s) = \text{K-Linear}_{\tau(s)}(h_s^{(k-1)})$, respectively. The matrix Multiplication in the parentheses takes the node types (Q-Linear$_{\tau(t)}$ and K-Linear$_{\tau(s)}$) and the edge type ($W_{\phi(e)}^A$) into account. Besides, the trainable prior variable $\mu_{\langle \tau(s), \phi(e), \tau(t) \rangle}$ plays the role of the adaptive scaling factor for each meta relation triplet $\langle \tau(s), \phi(e), \tau(t) \rangle$. After retrieving the unnormalized score, HGT conduct a softmax activation upon A ((k) and (s, e, t) are omitted, same below) for each $s \in \mathcal{N}(t)$ and get the attention score α of the current attention head. At last, all independent attention heads are concatenated, forming the heterogeneous mutual attention, *i.e.*, Attention = Concat($\alpha_1, \cdots, \alpha_h$).

Target-specific aggregation. After gathering the message and computing the attention score from all neighbors $s \in \mathcal{N}(t)$ of each node t, we can thus simply average the messages using the attention scores as the weights. The aggregation is formulated as below:

$$a_t^{(k)} = \sum_{s \in \mathcal{N}(t)} \left(\text{Attention}^{(k)}(s, e, t) \cdot \text{Message}^{(k)}(s, e, t) \right) \quad (5)$$
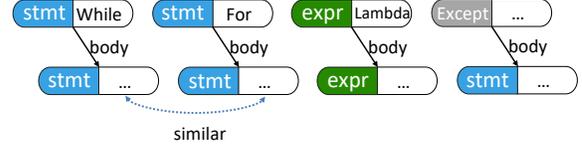
Figure 6: An illustrative example to demonstrate the effectiveness of the meta relations, which jointly uses the edge types and the node types. One may easily differ the meta relation of `stmt-body-stmt` from `Except-body-stmt`, and therefore the semantics of the body edges pointing to loop bodies and exception handlers can be distinguished.

Residual combination. Last but not the least, for each node t, the aggregated vector $a_t^{(k)}$ is combined with the residual information of t ($h_t^{(k-1)}$), producing the new representation of the current HGT layer. The combination is formulated as:

$$h_t^{(k)} = \sigma \left(\text{C-Linear}_{\tau(t)} \left(a_t^{(k)} \right) \right) + h_t^{(k-1)} \quad (6)$$

where σ is the activation function, and C-Linear$_{\tau(t)}$ also takes the node type of t ($\tau(t)$) into consideration.

5.2 Meta Relation in HGT

The HGT encoder leverages the types of the nodes and the edges jointly, as illustrated in the last section. During the message passing process, all Linear components and all W weights are associated with the types of the nodes or the edges (please refer to the $\tau(s)$, $\tau(t)$ and $\phi(e)$ subscripts in Eq. 3-5). In addition, the prior $\mu_{\langle \tau(s), \phi(e), \tau(t) \rangle}$ directly models the meta relation triplet. Therefore, in HGT, different sets of parameters are adopted in the message passing processes for various of meta relations determined by $\langle \tau(s), \phi(e), \tau(t) \rangle$.

Take Figure 6 for illustration, the edges are all with the same type body, but the semantics may differ from each other. *E.g.*, the body edges in For-body and While-body both indicates the body of the loop, while body in Except-body refers to an exception handler. If the types of the nodes and the edges are utilized independently by the model, it is hard to distinguish such differences among the semantics of the body edges. Fortunately, the meta relation explicitly suggests this kind of semantics differences, as one can easily tells the differences between `stmt-body-stmt` and `Except-body-stmt`. Therefore, HGT is capable of producing accurate and delicate code representations by modeling such meta relations jointly.

5.3 Downstream Module for HGT

HGT, serving as an encoder, produces a vectorized representation of the nodes and the whole graph. To complete a variety of downstream tasks, we need to employ the downstream modules. Most program processing tasks can be categorized into generation tasks and classification tasks. ❶ In generation tasks, such as method name prediction [2, 6], the model is supposed to output a sequence of information, *e.g.*, comment texts, method names, *etc.*, based on the context of the input program (code representation). We equip HGT with the pointer network for generation tasks, which can either generate new words or copy from the context directly. ❷ In classification tasks, such as code classification [30], the model makes classification based on the code representation. We connect an MLP classifier to HGT for classification tasks.

HGT for generation tasks. Generation tasks demand an accurate understanding of the semantics of the input code snippet. Therefore, we follow the classic encoder-decoder architecture [11], and utilize a transformer model [42] to decode based on the representations of the subtoken nodes in HPG from HGT. Besides direct decoding, we also incorporate the pointer network into our model. At each decoding step, the model operates in three steps. ❶ The model collects representations of the subtoken nodes $(s_1, \cdots, s_n)$ produced by HGT $(h_{s_1}, \cdots, h_{s_n})$, and conducts attention upon them (the attention scores are denoted as $a_1, \cdots, a_n$). In addition, the context vector h^* is also obtained as a weighted average, *i.e.*, $h^* = \sum_{i=1}^{n} a_i \cdot h_{s_i}$. ❷ Based on h^*, the model determines the probability of copy from the subtoken nodes (p_{copy}). On the other hand, the probability of generating a new word by the transformer is defined as $p_{gen} = 1 - p_{copy}$. ❸ The probability of decoding the word w is computed as: $Prob(w) = p_{gen} \cdot \text{Transformer}(w) + p_{copy} \cdot \sum_{i:s_i=w} a_i$, where Transformer$(w)$ is the probability that the transformer model outputs w and $\sum_{i:s_i=w} a_i$ is the probability that w is copied from the subtoken nodes. $Prob(w)$ is the joint probability of generating and copying w by the model. The copying mechanism empowered by the pointer network alleviates the OOV problem [43] of the transformer, by allowing the model to point directly to occurrences of the words. This can effectively improve the performance for those less frequent words during generation.

HGT for classification tasks. The classification tasks are rather simple, compared with the generation tasks. It involves two steps to complete the classification task with HGT. ❶ We must obtain the overall representation of the whole graph. We apply global attention pooling [26] over representations of all nodes in HPG. ❷ After receiving the graph representation, we employ an MLP classifier to conduct classification.

6 EVALUATION

With HPG and HGT, we perform in-depth evaluations upon four datasets against multiple current SOTA baselines to investigate the following research questions:

RQ1. On Generation Task. How does HPG+HGT perform on generation tasks? Specifically, on method name prediction, is HPG+HGT capable to outperform other SOTA approaches?

RQ2. On Classification Task. How does HPG+HGT perform on classification tasks? Specifically, how does our approach perform compared with the SOTA on the code classification task?

RQ3. Subtoken Scheme. Can the subtoken technique in HPG improve the performance? To what extent do the shared scheme and the independent scheme affect the performance of HGT?

RQ4. Ablation Study. Is the heterogeneous type information in HPG capable to boost the better representation of the GNN model? To what extent does HPG improve the model performance? What is the impact of the components in the design of HPG and HGT? How do they influence the performance of the downstream tasks?

6.1 Experiment Setting

The experiments are carried out upon one classic generation task (*i.e.*, method name prediction) and one classic classification task (*i.e.*,

Table 1: Statistics of the datasets

		Method Name Prediction		Code Classification	
		CSN-Python	Java-small	Python800	Java250
Size	Train	412,178	691,974	144,000	45,000
	Valid	23,107	23,844	48,000	15,000
	Test	22,176	57,088	48,000	15,000
# or len	Node	184.68	92.22	179.76	279.84
	Edge	684.21	246.77	650.14	894.56
	Name	2.2	2.5	-	-
Type	Node	17	77	17	77
	Edge	118	104	116	104

code classification). For each task, we evaluate upon two widely-studied datasets (four datasets in total). The baseline models for comparison are among the classic models or the SOTA models.

HPG parser. We implement HPG parsers for two popular programming languages – Python and Java. The Python parser conforms to the official Python 3.7 ASDL grammar[3]. Our implemented Python parser can extract up to 23 types of nodes and 71 types of edges (forward). As for Java, we implement an HPG parser based on tree-sitter[4], which parses the code snippet into the parsing tree. We manually define the field type for Java and assign types to the edges in the parsing tree according to our rules.

Subject task and dataset. We evaluate our approach on method name prediction for generation tasks and code classification for classification tasks. Specifically, for method name prediction, we select CodeSearchNet-Python (CSN-Python) [58] and Java-small [6, 58] datasets, and for code classification, we select Python800 and Java250 [33] datasets. For all these four benchmarks, we utilize the publicly available well-splitted datasets during evaluation.

In method name prediction, a method with its name masked is fed into the model, and the model needs to predict the original method name. **CSN-Python** originates from the CodeSearchNet corpus [21], consisting of around 450K real-world Python methods. **Java-small** is a widely-studied benchmark, containing 11 open-source Java projects from GitHub. We keep the data split of training, validation and testing consistent with existing work [6, 58]. Those examples that cannot be parsed by our HPG parser are discarded.

In code classification, the model needs to predict the category of the given code snippet. Both **Python800** and **Java250** comes from the CodeNet project [33]. The datasets are from two open judge platforms, and the snippets are categorized by the problem. The data split is consistent with the original authors[5], and we pre-process the samples with our HPG parsers. Table 1 summarizes the statistics of these datasets.

Baseline model. In method name prediction, we compare our proposed HPG+HGT with code2seq [6], GREAT [17], XLNet [52], Code Transformer [58], Sequence GNN [14], Sequence GINN [48] and Cognac [45]. As for code classification, the baselines includes GCN [24], GIN [51], GGNN [26], Tree-LSTM [41], TBCNN [30], TreeCaps

---

[3]https://docs.python.org/3.7/library/ast.html#abstract-grammar
[4]https://tree-sitter.github.io/tree-sitter/
[5]The dataset split we use in our experiments is the same as stated by the authors which is provided in https://github.com/IBM/Project_CodeNet/issues/29

Table 2: Performance of models on CSN-Python (in %).

Model	Precision	Recall	F1	p-value[†]
GCN	24.77	19.97	22.12	< .0001
GGNN	24.07	19.09	21.29	< .0001
code2seq [6]	35.79	24.85	29.34	< .0001
GREAT [17]	35.07	31.59	33.24	< .0001
XLNet [52]	37.39	32.01	34.49	< .0001
Code Transformer [58]	36.41	**33.68**	34.99	< .0001
Cognac [45]	32.71	27.63	29.96	< .0001
HPG+HGT (no_subtoken)	46.48	26.61	33.84	< .0001
HPG+HGT (independent)	**48.44**	29.56	**36.71**	-
HPG+HGT (shared)	47.69	28.49	35.67	< .0001

[†] P-values are calculated by comparing with HPG+HGT (independent).

[9]) and GREAT [17]. Since CodeNet has already provided simplified parse trees (SPT) for benchmarks, we directly use them for training and evaluation. Our selected baseline models contain both classic code representation models (GCN and TBCNN, *etc.*) and SOTA solutions (GREAT and Code Transformer, *etc.*). The required data formats of the baselines cover token sequences (XLNet and Cognac), AST paths (code2seq), ASTs (Tree-LSTM, TBCNN, and TreeCaps), and graphs (GCN, GIN, GGNN, Sequence GINN, Code Transformer, and GREAT). The architectures of the baselines include RNNs (code2seq and Cognac), convolutional models (TBCNN and TreeCaps), recursive models (Tree-LSTM), GNNs (GCN, GIN and GGNN, Sequence GINN), and transformers (XLNet, Code Transformer, and GREAT).

Performance indicators. In method name prediction, we list subtoken-level precision, recall, and F1-score over the target sequence (case insensitive) as the indicator, following the previous work [6, 7, 31]. Precision measures the accurate subtoken ratio in the prediction, recall measures the hitting ratio in the target, and F1-score balances precision and recall. *E.g.*, supposing the target is $\{'train', 'graph', 'model'\}$ and the prediction is $\{'train', 'model'\}$, then the precision is 1.0, the recall is $\frac{2}{3}$, and the F1-score is 0.8 As for code classification, we employ accuracy as the major performance indicator, as CodeNet suggests [33].

A/B testing. We also conduct A/B Testing [25] to show the significance of the experimental results. Specifically, we employ A/B testing to decide whether HPG+HGT outperforms the other baselines with confidence scores.

We implement all models with the PyTorch framework along with the DGL library [6] and train them on an NVIDIA Tesla V100 16GB GPU. In our experiment, the HGT encoder consists of 8 HGT layers. We set the embedding size and the hidden size to 256 and 1,024, respectively. We employ 8 heads in each HGT layer. We set the dropout rate to 0.2. We adopt the AdamW [29] optimizer with the learning rate of 1×10^{-4}. All models are trained from scratch.

6.2 RQ1: On Generation Task

To answer RQ1, we evaluate HPG+HGT for generation tasks, compared with Code Transformer, Cognac, along with other baselines, upon CSN-Python and Java-small. We list the results of HPG+HGT

---
[6] https://www.dgl.ai/

Table 3: Performance of models on Java-small (in %).

Model	Precision	Recall	F1	p-value[†]
Sequence GINN [48]	64.8	56.2	60.2	< .0001
Sequence GNN [14]	–	–	51.3	< .0001
GGNN	40.3	35.3	36.9	< .0001
code2seq [6]	50.6	37.4	43.0	< .0001
GREAT [17]	53.6	46.4	49.8	< .0001
XLNet [52]	55.5	46.1	50.3	< .0001
Code Transformer [58]	54.9	49.8	52.2	< .0001
Original Cognac [45] [‡]	67.1	59.7	63.2	–
Cognac [45]	–	–	57.7	< .0001
HPG+HGT (no_subtoken)	62.2	56.1	59.1	< .0001
HPG+HGT (independent)	**65.3**	**57.2**	**61.0**	-
HPG+HGT (shared)	64.7	56.1	60.1	< .0001

[†] P-values are calculated by comparing with HPG+HGT (independent).
[‡] Cognac incorporates additional Java-specific prior knowledge. We list the original results here, but we do not compare HPG+HGT with it.

along with the baselines upon CSN-Python and Java-small in Table 2 and 3, respectively.

CSN-Python. From Table 2, we find that HPG+HGT outperforms the baseline models on the precision and F1-score indicators. Especially, HPG+HGT with independent subtokens outperforms the current SOTA Code Transformer by 1.72%, and it outperforms all baselines significantly with high confidence (p-value< 0.0001).

There are also some exceptions in Table 2, as the recall indicator of HPG+HGT is lower than the current SOTA GREAT, XLNet, and Code Transformer models. Our in-depth investigation on the intermediate log reveals that HPG+HGT tends to predict "short" method names with only a few subtokens, while the other models prefer to produce "long" method names. This may explain why HPG+HGT has a lower recall, as recall reflects the hit rate of the prediction in the ground-truth, regardless of the predicted name length. This explanation is consistent with the phenomenon where the baseline models have lower precision – precision measures the ratio of the accurate prediction, and hence the baselines preferring long method names perform poorly on this indicator.

Java-small. HPG+HGT outperforms the baseline models greatly upon Java-small, as presented in Table 3. Specially, HPG+HGT with independent subtokens even outperforms most baselines by more than 10% on F1-score (except for GINN and Cognac). The performance improvement is significant with high confidence (p-value< 0.0001)

In addition, we must elaborate on some technical details of the Cognac [45] model. The full Cognac model leverages additional caller-callee information, which is specially designed for the Java programming language alone. However, unlike the heterogeneous types in HPG, for other programming languages, such additional information in Cognac is hardly available. This can explain why Cognac performs poorly upon CSN-Python, which is Python rather than Java. To demonstrate the general performance of HPG+HGT for generation tasks in general scenarios, we employ the trivial Cognac model as the baseline during evaluation, *i.e.*, the caller-callee information is not provided for both datasets, even though the performance of the original Cognac is also listed in Table 3.

Table 4: Performance of models for code classification (in %).

Model	Python800		Java250	
	Acc	p-value†	Acc	p-value†
GCN	91.81	< .0001	89.06	< .0001
GIN	93.17	< .0001	90.76	< .0001
GGNN	89.92	< .0001	88.46	< .0001
Tree-LSTM (root) [41]‡	93.95	< .0001	93.19	< .0001
Tree-LSTM (attention)‡	93.83	< .0001	93.71	0.0036
TBCNN [30]	91.10	< .0001	90.32	< .0001
TreeCaps [9]	90.26	< .0001	91.42	< .0001
GREAT [17]	93.30	< .0001	93.15	< .0001
HPG+HGT (no_subtoken)	93.81	< .0001	93.45	0.0004
HPG+HGT (independent)	94.35	< .0001	93.59	0.0021
HPG+HGT (shared)	**94.99**	-	**93.95**	-

† P-values are calculated by comparing with HPG+HGT (shared).
‡ Two Tree-LSTM variants: root representation and global attention pooling.

> **Answer to RQ1:** Our proposed approach generally outperforms the current SOTA baseline models on CSN-Python and Java-small method name prediction tasks. It suggests that HPG+HGT may be capable of producing more accurate and delicate code representations for the generation tasks.

6.3 RQ2: On Classification Task

To answer this RQ, we compare the performance of HPG+HGT with the tree-based and graph-based baselines upon the Python800 and Java250 classification tasks.

We present the classification accuracies of HPG+HGT and other baseline models on Python800 and Java250 in Table 4. In most cases, HPG+HGT with the subtoken technique outperforms the baseline models by at least 2% on both datasets. Tree-LSTM performs comparably with HPG+HGT, but HPG+HGT still outperforms it by 1.04% and 0.24% on Python800 and Java250, respectively. The p-values (in most cases < 0.0001) also demonstrate the significant improvement of HPG+HGT compared with the baselines.

We also discover an interesting phenomenon – the shared subtoken scheme is better for code classification (see Table 4), while the independent scheme is more effective for method name prediction (see Table 2 and 3). We will discuss it in **RQ3**.

> **Answer to RQ2:** Our proposed approach HPG+HGT outperforms all the tree-based and graph-based baselines upon Python800 and Java250 datasets. It suggests that our proposed heterogeneous-graph-based approach may extract program semantics and handle the classification tasks well.

6.4 RQ3: Subtoken Scheme

To investigate the impact of suchtoken scheme, as we have found earlier, we compare the HPG+HGT models with different subtoken schemes, *i.e.*, the shared scheme, the independent scheme, and the no-subtoken scheme, upon all four datasets.

Impact of subtoken. The subtoken nodes may be beneficial for HPG+HGT in both generation and classification tasks. In method

Table 5: Performance of GGNN and HGT (in %) with and without heterogeneous type information on CSN-Python.

Model	Precision	Recall	F1	ΔF1
GGNN	24.07	19.09	21.29	–
+ edge type	27.31	21.94	24.33	↑ 3.04
HPG+HGT	48.44	29.56	36.71	–
- node type	43.55	21.80	29.06	↓7.65
- edge type	43.57	22.58	29.74	↓6.97
- node type & edge type	43.03	21.44	28.62	↓8.09

name prediction, HPG+HGT with subtoken (both the shared scheme and the independent scheme) performs better than HPG+HGT without subtoken, as Table 2 and 3 indicates. Specifically, by adopting the independent subtoken scheme, the F1-score of HPG+HGT increases at least 1.9% upon both CSN-Python and Java-small datasets. As for code classification, similar findings can be drawn according to Table 4. Specially, HPG+HGT with shared subtoken scheme outperforms HPG+HGT without subtoken by about 1.2% on Python800.

Subtoken scheme for generation task. The independent scheme may be more effective for method name prediction than the shared scheme, as in Table 2 and 3, HPG+HGT with the independent scheme outperforms the shared scheme. There are three plausible reasons for this phenomenon. ❶ The shared scheme may break the order of the subtoken nodes in the original snippet, which can be informative for generation tasks. ❷ For those subtokens frequently appear in the code snippet, the corresponding subtoken nodes occur only once in HPG with the shared subtoken scheme. However, the copy mechanism (the pointer network) in the decoder takes the repeated subtokens into account, and the shared scheme may be unfavorable. ❸ The independent scheme may model the polysemous subtokens, *i.e.*, it allows different representations for the subtoken nodes with the same values. This property is likely to be advantageous to generation tasks.

Subtoken scheme for classification task. Unlike method name prediction, for code classification, the shared scheme may be favorable than, as presented in Table 4. Specifically, HPG+HGT with the shared scheme slightly outperforms the independent scheme by about 0.7% and 0.4% upon Python800 and Java250, respectively. We assume that the shared subtoken nodes may reflect the association among the identifiers in the snippet, which can boost the performance for classification tasks.

> **Answer to RQ3:** The subtoken technique is capable to improve the performance of HGT for both generation and classification tasks. Different types of tasks may benefit from different subtoken schemes. Specifically, the independent scheme may be good for method name prediction, while the shared scheme may benefit code classification.

6.5 RQ4: Ablation Study

Ablation studies are carried out to illustrate the role of the components playing in HPG+HGT, including the heterogeneous types, the manually crafted edges in HPG, and the decoding strategies. Specifically, we include the additional edge types from HPG to verify the performance gaining upon the classic GGNN model, and we remove
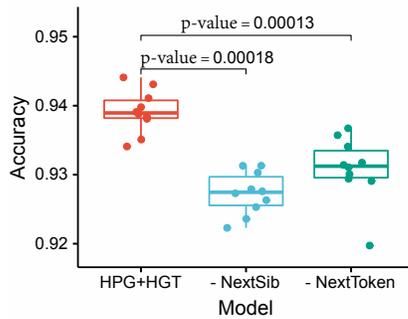
Figure 7: Impact of removing the manually crafted `NextSib` and `NextToken` edges to HPG+HGT upon Java250 (in %) with p-values.

the types of the nodes and the edges to examine the performance loss of our proposed HPG+HGT. We also carry out ablations by removing the manually crafted `NextSib` and `NextToken` edges, and adopting different decoding strategies.

Impact of heterogeneity. To demonstrate the impact of heterogeneous types in HPG, we evaluate GGNN and HPG+HGT upon both homogeneous and hetgerogeneous graphs for CSN-Python, as listed in Table 5. ΔF1 refers to the change of the absolute value of the F1-score indicator compared with the original configuration. We add additional edge types (from HPG) to GGNN, which originally processes the homogeneous graph [7]. Table 5 indicates that with the help of the additional edge types, the performance of GGNN significantly improves. Especially, the F1-score rises about 3%. Therefore, GGNN may benefit from the heterogeneous graph. On the other hand, we remove the node types or the edge types or both from HPG, to show the performance loss of HGT when the heterogeneous type information is removed from HPG. We are able to conclude from Table 5 that either removing the node types or the edge types from HPG would cause the F1-score to drop about 7%. Furthermore, when both the edge and the node types are removed, *i.e.*, the graph degenerates to be homogeneous, HGT loses performance greatly, as the F1-score drops 8%. Therefore, the heterogeneous types in HPG can be quite essential for HGT to generate accurate code representations for tasks such as CSN-Python.

Manually crafted edge. To demonstrate the usefulness of the manually crafted `NextSib` and `NextToken` edges in HPG, we perform an ablation by removing them from HPG. Figure 7 presents the accuracy loss of HPG+HGT with shared subtokens on Java250 when the edges are removed. We evaluate the three HGT models for 10 times upon subsets of the testing set. The accuracy drops about 1.4% and 1.0% when `NextSib` and `NextToken` are removed respectively. The p-values (< 0.0002) indicate that the full HPG+HGT model is significantly better than those without `NextSib` or `NextToken` edges upon Java250. This ablation demonstrates the effectiveness of the manually crafted edges in HPG, and may provide insight into the design of the program graph.

Decoding strategy. As introduced in section 5.3, HGT for generation decodes over only the `subtoken` nodes. To prove our design,

Table 6: Performance of HPG+HGT with different decoding strategies on CSN-Python (in %).

Decoding strategy	Precision	Recall	F1	ΔF1
Over subtoken nodes	48.44	29.56	36.71	–
Over all nodes	40.55	26.63	32.14	↓4.57

we conduct another ablation upon HPG+HGT with independent subtokens on CSN-Python, where the decoding strategy over all node representations is evaluated. As table 6 indicates, decoding over all nodes may cause the performance to decrease greatly – the F1-score decreases about 4.6% compared with the original design. To some extent, it verifies our design for generation tasks of the decoding strategy in HPG+HGT.

> **Answer to RQ4:** Ablation demonstrates the usefulness of the heterogeneous types in HPG. It also confirms the effectiveness of our design for the manually crafted edges in HPG and the decoding strategy over only the `subtoken` nodes in HGT.

6.6 Threats to Validity

The dataset selection could be a threat to validity. We counter this by selecting classic method name prediction for generation tasks and code classification for classification tasks. We further select four datasets, which are widely adopted in previous work. Another threat could be the baseline selection. We counter this by comparing our proposed HPG+HGT with both the classic and the SOTA models. A further threat could be the language support. We counter this by implementing the HPG parser for the popular Python and Java languages. Our in-depth evaluation, costing more than 3 weeks and over 500 GPU hours, has already demonstrated the effectiveness of HPG+HGT. Since the ASDL grammar is open-sourced, all it takes to build HPG parsers for other programming languages is a large amount of engineering effort. We leave the supporting to more programming languages in future work.

7 CONCLUSION

In this paper, we put forward the idea of heterogeneity in AST and present a framework of representing source code as heterogeneous program graphs (HPG) using the ASDL grammar. By applying heterogeneous graph transformer (HGT) on our proposed HPG, our approach is capable to generate accurate and delicate representations for programs. Our in-depth evaluations on four classic datasets for two typical code processing tasks (*i.e.*, method name prediction and code classification) demonstrate that the heterogeneous type information in our proposed HPG improves the performance of the representation model. Furthermore, our proposed HPG+HGT solution even outperforms the SOTA baselines for generation and classification tasks.

8 ACKNOWLEDGEMENT

---

[7] The GGNN model can leverage the edge type during the message passing process. For each node, GGNN can use the edge-type dependent weight to aggregate the features from its neighbors.

REFERENCES

[1] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A Transformer-based Approach for Source Code Summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. 4998–5007.

[2] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2015. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, Elisabetta Di Nitto, Mark Harman, and Patrick Heymans (Eds.). ACM, 38–49. https://doi.org/10.1145/2786805.2786849

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

[4] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *International Conference on Learning Representations*.

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

[6] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating Sequences from Structured Representations of Code. In *International Conference on Learning Representations*.

[7] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: learning distributed representations of code. *Proc. ACM Program. Lang.* 3, POPL (2019), 40:1–40:29. https://doi.org/10.1145/3290353

[8] Marc Brockschmidt, Miltiadis Allamanis, Alexander L Gaunt, and Oleksandr Polozov. 2019. Generative Code Modeling with Graphs. In *International Conference on Learning Representations*.

[9] Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. 2021. TreeCaps: Tree-Based Capsule Networks for Source Code Processing. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*. AAAI Press, 30–38. https://ojs.aaai.org/index.php/AAAI/article/view/16074

[10] Ruichu Cai, Zhihao Liang, Boyan Xu, Zijian Li, Yuexing Hao, and Yao Chen. 2020. TAG : Type Auxiliary Guiding for Code Comment Generation. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. 291–301.

[11] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, Alessandro Moschitti, Bo Pang, and Walter Daelemans (Eds.). ACL, 1724–1734. https://doi.org/10.3115/v1/d14-1179

[12] Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. 2020. Hoppity: Learning Graph Transformations to Detect and Fix Bugs in Programs. In *International Conference on Learning Representations*.

[13] David Duvenaud, Dougal Maclaurin, Jorge Aguilera-Iparraguirre, Rafael Gómez-Bombarelli, Timothy Hirzel, Alán Aspuru-Guzik, and Ryan P. Adams. 2015. Convolutional Networks on Graphs for Learning Molecular Fingerprints. In *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, Corinna Cortes, Neil D. Lawrence, Daniel D. Lee, Masashi Sugiyama, and Roman Garnett (Eds.). 2224–2232. https://proceedings.neurips.cc/paper/2015/hash/f9be311e65d81a9ad8150a60844bb94c-Abstract.html

[14] Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Structured Neural Summarization. In *International Conference on Learning Representations*.

[15] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. 2017. Neural Message Passing for Quantum Chemistry. In *International Conference on Machine Learning*. 1263–1272.

[16] Marco Gori, Gabriele Monfardini, and Franco Scarselli. 2005. A new model for learning in graph domains. In *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, Vol. 2. IEEE, 729–734.

[17] Vincent J Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. 2020. Global relational models of source code. In *International Conference on Learning Representations*.

[18] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Comput.* 9, 8 (1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

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

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

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

[22] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing Source Code using a Neural Attention Model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics. https://doi.org/10.18653/v1/p16-1195

[23] Lin Jiang, Hui Liu, and He Jiang. 2019. Machine Learning Based Recommendation of Method Names: How Far are We. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 602–614. https://doi.org/10.1109/ASE.2019.00062

[24] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *International Conference on Learning Representations*.

[25] Ron Kohavi and Roger Longbotham. 2017. Online Controlled Experiments and A/B Testing. *Encyclopedia of machine learning and data mining* 7, 8 (2017), 922–929.

[26] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. 2016. Gated Graph Sequence Neural Networks. In *International Conference on Learning Representations*, Yoshua Bengio and Yann LeCun (Eds.).

[27] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2021. A Context-based Automated Approach for Method Name Consistency Checking and Suggestion. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 574–586. https://doi.org/10.1109/ICSE43902.2021.00060

[28] Kui Liu, Dongsun Kim, Tegawendé F. Bissyandé, Tae-young Kim, Kisub Kim, Anil Koyuncu, Suntae Kim, and Yves Le Traon. 2019. Learning to spot and refactor inconsistent method names. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 1–12. https://doi.org/10.1109/ICSE.2019.00019

[29] Ilya Loshchilov and Frank Hutter. 2019. Decoupled Weight Decay Regularization. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. https://openreview.net/forum?id=Bkg6RiCqY7

[30] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, Dale Schuurmans and Michael P. Wellman (Eds.). AAAI Press, 1287–1293. http://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/11775

[31] Son Nguyen, Hung Phan, Trinh Le, and Tien N. Nguyen. 2020. Suggesting natural method names to check name consistencies. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 1372–1384. https://doi.org/10.1145/3377811.3380926

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

[33] Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. 2021. CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*. https://openreview.net/forum?id=6vZVBkCDrHT

[34] Jiezhong Qiu, Jian Tang, Hao Ma, Yuxiao Dong, Kuansan Wang, and Jie Tang. 2018. Deepinf: Social influence prediction with deep learning. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2110–2119.

[35] Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract Syntax Networks for Code Generation and Semantic Parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*. 1139–1149.

[36] Tushar Sharma, Maria Kechagia, Stefanos Georgiou, Rohit Tiwari, and Federica Sarro. 2021. A Survey on Machine Learning Techniques for Source Code Analysis. *CoRR* abs/2110.09610 (2021). arXiv:2110.09610 https://arxiv.org/abs/2110.09610

[37] Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. 2018. 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)*. 464–468.

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

[39] Yizhou Sun and Jiawei Han. 2012. *Mining Heterogeneous Information Networks: Principles and Methodologies*. Morgan & Claypool Publishers. https://doi.org/10.2200/S00433ED1V01Y201207DMK005

[40] Jeffrey Svajlenko, Judith F. Islam, Iman Keivanloo, Chanchal Kumar Roy, and Mohammad Mamun Mia. 2014. Towards a Big Data Curated Benchmark of Inter-project Code Clones. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014.*

IEEE Computer Society, 476–480. https://doi.org/10.1109/ICSME.2014.77

[41] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. 2015. Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks. *CoRR* abs/1503.00075 (2015). arXiv:1503.00075 http://arxiv.org/abs/1503.00075

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

[43] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. 2015. Pointer networks. In *Advances in neural information processing systems*. 2692–2700.

[44] Daniel C. Wang, Andrew W. Appel, Jeffrey L. Korn, and Christopher S. Serra. 1997. The Zephyr Abstract Syntax Description Language. In *Proceedings of the Conference on Domain-Specific Languages, DSL'97, Santa Barbara, California, USA, October 15-17, 1997*, Chris Ramming (Ed.). USENIX, 213–228. http://www.usenix.org/publications/library/proceedings/dsl97/wang.html

[45] Shangwen Wang, Ming Wen, Bo Lin, and Xiaoguang Mao. 2021. Lightweight global and local contexts guided method name recommendation with prior knowledge. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 741–753. https://doi.org/10.1145/3468264.3468567

[46] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. 2020. 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)*. IEEE, 261–271.

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

[48] Yu Wang, Ke Wang, Fengjuan Gao, and Linzhang Wang. 2020. Learning semantic program embeddings with graph interval neural network. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 137:1–137:27. https://doi.org/10.1145/3428205

[49] Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. 2019. LambdaNet: Probabilistic Type Inference using Graph Neural Networks. In *International Conference on Learning Representations*.

[50] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. 2021. A Comprehensive Survey on Graph Neural Networks. *IEEE Trans. Neural Networks Learn. Syst.* 32, 1 (2021), 4–24. https://doi.org/10.1109/TNNLS.2020.2978386

[51] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How Powerful are Graph Neural Networks?. In *International Conference on Learning Representations*.

[52] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime G. Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. 2019. XLNet: Generalized Autoregressive Pretraining for Language Understanding. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 5754–5764. https://proceedings.neurips.cc/paper/2019/hash/dc6a7e655d7e5840e66733e9ee67cc69-Abstract.html

[53] Pengcheng Yin and Graham Neubig. 2018. 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*. 7–12.

[54] Pengcheng Yin, Graham Neubig, Miltiadis Allamanis, Marc Brockschmidt, and Alexander L Gaunt. 2019. Learning to Represent Edits. In *International Conference on Learning Representations*.

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

[56] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 783–794. https://doi.org/10.1109/ICSE.2019.00086

[57] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Advances in Neural Information Processing Systems*. 10197–10207.

[58] Daniel Zügner, Tobias Kirschstein, Michele Catasta, Jure Leskovec, and Stephan Günnemann. 2021. Language-Agnostic Representation Learning of Source Code from Structure and Context. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net. https://openreview.net/forum?id=Xh5eMZVONGF