SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

July 2025, Vol. 68, Iss. 7, 172101:1–172101:21
https://doi.org/10.1007/s11432-023-3898-5

Learning to represent code semantics

Fang LIU[1,2], Ge LI[3,4], Qianhui ZHAO[1,2] & Li ZHANG[1,2*]

[1]*State Key Laboratory of Complex & Critical Software Environment, Beijing 100191, China*
[2]*School of Computer Science & Engineering, Beihang University, Beijing 100191, China*
[3]*Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, Beijing 100871, China*
[4]*School of Computer Science, Peking University, Beijing 100871, China*

Received 26 May 2023/Revised 9 August 2023/Accepted 13 October 2023/Published online 19 June 2025

Abstract Code semantic learning serves as the basis of many program analysis tasks. Researchers have paid much effort to build robust and effective code representation models over the years. One line of work focuses on introducing the code structure into the representations. To further improve the robustness of the code representation, approaches based on compiler intermediate representations (IRs) are proposed. However, these IR-based models suffer from heavy computational costs and memory overhead. How to represent program semantics effectively and efficiently still remains a challenge. To this end, we propose EECS, an effective and efficient code semantic representation approach based on compiler IRs and a hybrid attention mechanism. For input representation, to address the unlimited vocabulary size issue in IR, we propose a variable identification strategy to allocate each register variable to a new ID that can represent their relative positions. Besides, we also extract the data flow information among the code blocks. Then we build a hierarchical multi-layer Transformer encoder to capture the data dependency information as well as the code semantics through a hybrid attention mechanism. To enable EECS to learn code semantics and functionality better, we optimize three objectives jointly during the training process. Experimental results on three code semantic understanding tasks show that EECS performs better than the state-of-the-art techniques, demonstrating the remarkable capability of EECS on program semantics understanding.

Keywords software engineering, code semantic learning, compiler intermediate representation, data dependency modeling, artificial intelligence

Citation Liu F, Li G, Zhao Q H, et al. Learning to represent code semantics. Sci China Inf Sci, 2025, 68(7): 172101, https://doi.org/10.1007/s11432-023-3898-5

1 Introduction

Code representation learning has attracted increasing attention in both software engineering and artificial intelligence fields, which aims to preserve code semantics in continuously distributed vectors from various code resources. It serves as a basis for many downstream tasks, such as code clone detection [1, 2], code summarization [3, 4], program classification [5, 6], and method name recommendation [7, 8]. To better understand the semantics, various code representation models are proposed for modeling code of different formats. Prior studies mainly represent code as a lexical token sequence due to its simplicity and feasibility, which is similar to natural language processing (NLP). Thus, many NLP representation learning techniques are adapted for modeling code token sequences [9, 10]. Inspired by the success of pre-trained models in the NLP field [11, 12], lots of studies pay attention to adapting the pre-training techniques for code representation learning, for example, CodeBERT [13], CodeT5 [14], and PLBART [15]. These pre-trained models have shown impressive performance in code semantic learning. However, the inconsistent input forms of the pre-training and downstream tasks always hinder performance, where the knowledge of pre-trained models might not be fully exploited. Moreover, the performance of fine-tuning heavily relies on the training corpus of the downstream tasks, and the performance of the data-scarce tasks is poor [16].

Besides, most of the above representations are based on the naturalness hypothesis [17] and adopt the straightforward token sequence modeling, focusing more on the naming and coding conventions. Thus are not sufficient enough for grasping the code functionality. For example, semantically equivalent

* Corresponding author (email: lily@buaa.edu.cn)

code snippets can be different in formats (via different programming languages (PLs), syntax structures, operations, and statement orders). On the other hand, code snippets that look similar can have different semantics (if $a > b$: vs. if $a < b$:).

To capture the semantic properties of programs more thoroughly, researchers began to incorporate code structure into representations using abstract syntax trees (ASTs) [18, 19], control/data flow graphs [20, 21], or augmenting AST with data dependency edges [22, 23]. Despite considering the structural information, these models are still sensitive to stylistic changes, which are inadequate to understand program semantics robustly. To build a semantically robust code representation model, approaches based on compiler intermediate representation (IR) are proposed. IR is used by a compiler or virtual machine to represent code semantics through an instruction set, which is language-agnostic. LLVM[1)] is a successful and widely used modern compiler, the IR generated by LLVM has many superior characteristics. For example, it provides easy access to control and data flow analysis and can support many lightweight run-time and inter-procedural optimizations. LLVM IR has been used for many code representation work [24–26]. They transform the IR instructions into graphs to capture the semantic relations between instructions and operands. Compared with the source code statements, the number of IR instructions is much bigger. After converting the IR to graph[2)], the number of verticals and edges is enormous. For example, in ProGraML [26], the average number of verticals per IR of the TensorFlow dataset is 5786, and the average number of edges per IR is 11482. After the graph is constructed, they built graph neural network (GNN)-based models to process it, resulting in high computational cost and memory overhead due to the big graph size and the high model complexity.

How to represent program semantics effectively and efficiently still remains a challenge, reflected in how to represent them and how to process them. To this end, we propose an effective and efficient code semantic representation approach (EECS) based on LLVM IR and a hybrid attention mechanism. Specifically, we represent code with LLVM IR, which is more robust for learning code semantics compared to high-level PLs.

For input representation, to reduce the vocabulary size as well as the model complexity, we propose a variable (register) identification strategy to allocate each variable to a new ID that can represent their relative positions. To efficiently capture the semantic information from the input IR data, we build a hierarchical multi-layer Transformer encoder. The lower-level sentence-BERT encoder is responsible for producing the vector representation for each IR instruction, aiming at reducing the sequence length of the input IR instructions and further reducing the time and space cost. Then we extract the data dependency information among code blocks, and build an instruction-level encoder to capture the data dependency information as well as the code semantics through a hybrid attention mechanism. During training, to enable EECS to learn code semantics and functionality more robustly, we train EECS with three code semantic learning objectives jointly, including (1) program classification, (2) identifying functionally equivalent optimization IR forms via contrastive learning, and (3) code semantic recovering (SR).

The contribution of this paper can be summarized as follows.

- We design an efficient and effective code semantic representation approach based on the LLVM IR and a hybrid attention mechanism.
- We propose three training objectives to enable the model to learn code semantics and functionality more robustly.
- We demonstrate the remarkable capability of EECS on program semantics understanding by the evaluation of three code semantic understanding tasks.

2 Preliminaries & research question formulation

In this section, we first introduce the background knowledge that is related to this work, including code representation learning and IR. Then we present an overview of the research questions (RQs).

2.1 Code representation learning

Code representation learning, also termed code embedding learning, aims to preserve code semantics in continuously distributed vectors from various code resources. It serves as a basis for current deep learning-based program analysis. Current code representation learning approaches fall into three categories:

1) https://llvm.org/.

2) The vertical consists of instructions, variables, and constants. The edges between verticals represent the flow information.

lexical-based, syntax-based, and semantic-based. Lexical-based code representation approaches represent the code based on its textual token sequences, which can reflect the lexical information [9,10]. To model the syntactic information of the code, syntax-based code representation approaches focus on incorporating the structural information into the code representation learning process by representing the AST using neural networks [5,18,19]. To further capture the semantic information, semantic-based approaches are proposed to represent the control and data flow dependencies using structural neural networks. One line of work extends ASTs through adding data/control flow edges to build the program graph [22,23], and then leverage GNNs for learning the representation of the graph. Another line of work represents the semantic information using the IRs [24,26]. They transform the IR instructions into graphs to capture the semantic relations between instructions and operands, and then build GNN-based models to process the graph.

Inspired by the success of pre-trained models in the NLP field [11,12], many pre-training techniques have been adapted for PLs, where the code representation is produced through learning from a large number of code resources, covering different code views, i.e., code tokens, ASTs, control/data flow graphs, and natural language descriptions (code comments). CodeBERT [13] is a bimodal pre-trained model for modeling both PL and natural language (NL). GraphCodeBERT [27] further incorporates the data-flow information into the model, aiming at capturing the code semantics better. Later, Guo et al. [28] proposed UniXcoder, which is a unified code representation model that incorporates semantic and syntax information from code comment and AST. Wang et al. [29] proposed SynCoBERT, a syntax-aware multi-modal code representation model. It is pre-trained with identifier prediction (IP) and AST edge prediction (TEP) objectives, aiming at capturing the symbolic and syntactic properties of source code. They also utilize a multi-modal contrastive learning strategy to exploit the complementary information in semantically equivalent among different modalities. Inspired by BART [30], Ahmad et al. [15] proposed PLBART, an encoder-decoder model that is pre-trained via denoising auto-encoding objective. It achieves advanced performance on various code-related tasks. Inspired by T5 [12], Wang et al. [14] proposed CodeT5, which is a unified pre-trained encoder-decoder model that can support both code understanding and generation tasks. It is pre-trained with identifier-aware and bimodal dual-generation pre-training objectives aiming at learning the token type and cross-modal alignment information. It has achieved state-of-the-art performance in many code understanding and generation tasks.

2.2 IR

As the name suggests, IR is a representation of a program between the source and target languages. It is the intermediate form of the program that is being compiled, and is the central data structure in a compiler. During the compile process, the compiler first parses and translates source code to IR, then generates the target code based on IR. The IR can preserve the code semantics, and is independent of both the source language and the target machine. LLVM[3)] is a successful and widely used modern compiler, the IR generated by LLVM has many superior characteristics. For example, LLVM IR is in static single assignment (SSA) form, where any local variable is assigned only once. It provides easy access to control and data flow analysis and can support many lightweight runtime and interprocedural optimizations. The source code of many general PLs can be easily compiled to LLVM IR.

To model code semantics in a robust manner, previous research has attempted to represent code based on IR instructions. Ben-Nun et al. [24] proposed a code representation learning approach NCC, which incorporates data flow and control flow information into LLVM-IR. They combined the proposed embedding with LSTM architecture and achieved superior performance in several code semantic learning tasks. Similarly, VenkataKeerthy et al. [31] proposed IR2VEC, which is a code embedding learning approach based on LLVM IR and data flow information. They use non-sequential models to train the embedding, resulting in faster training time. Different from the above work, ProGraML [26] built a graph based on the compiler IRs, and then used the GNN to learn the representation. Compared with the prior approaches, ProGraML is able to capture control, data, and call relations, and thus is more expressive. It outperforms prior state-of-the-art approaches in several traditional compiler analysis tasks as well as two high-level code semantic learning tasks. Nonetheless, it shows a lot of computational cost and memory overhead due to the big graph size and the high model complexity. How to represent the code semantics effectively and efficiently still remains a challenge.

3) https://llvm.org/

Table 1 Statistical analysis results on the uniqueness and sizes of IR files.

	Tokens	Instructions
Total number	1535258675	191972335
Unique number	1669108	5078182
Average number per file	1154	409
Average number per block	66	8
Average occurrence	38	920
# of occurrence >1	0.55	0.44

2.3 RQs

We seek to answer the following RQs to evaluate the effectiveness of EECS.

RQ1: Effectiveness and practicality in semantic understanding. How effective is our approach in code semantic understanding? Is it practical enough for real-world applications?

RQ2: Generalization capability analysis. To what extent does our approach fare in semantic understanding scenarios involving unseen languages?

RQ3: Ablation study. How do different components in EECS contribute to performance improvements?

RQ4: Sensitivity analysis. How do various factors affect EECS's performance, such as IR lengths and training data size?

In RQ1, we choose two typical code semantic understanding tasks, i.e., algorithm classification and heterogeneous compute device mapping, to validate the effectiveness and practicality of our approach, and compare it with several advanced program representation approaches. EECS is trained on the IRs generated by the C++ language. In RQ1, EECS is also evaluated on the C language family (C++, OpenCL). To evaluate the generalization capability of EECS, in RQ2, we perform a new experiment on the Java language. Specifically, we choose Java to C++ program translation task, that is, given a Java method (with its LLVM IR code), the model is used to translate the function into a method written in C++ language. We evaluate the model's performance on the GeeksforGeeks program translation dataset provided by TransCoder [32]. In RQ3, we conduct an ablation study to analyze the effectiveness of each component in EECS. Finally, we investigate the sensitivity of our approach to IR lengths and training data size in RQ4, aiming to evaluate the influence of these choices on the performance of EECS.

3 Statistical analysis and motivation

To confirm our standpoint about the enormous number of IR instructions in the previous section, we perform a statistical analysis on the uniqueness and size of IRs from the perspective of the token number and the instruction number. We use the DeepDataFlow datasets[4)] proposed by Cummins et al. [26], which consists of 250428 IR files from popular open-source repositories that cover a diverse range of domains and source languages. Each source file is converted to LLVM-IR[5)].

3.1 Uniqueness and sizes of IR instructions

To analyze the uniqueness and sizes of IR instruction tokens, we perform statistical analysis on the IR files of the dataset. The results are shown in Table 1. According to the results, from the perspective of instructions, there are 5078182 unique instructions among all 191972335 instructions, accounting for about 3%. From the perspective of tokens, there are 1669108 unique tokens among 1535258675, accounting for about 0.1%. As seen from the third row, for each IR file, there are more than one thousand tokens on average, but only about four hundred statements. Figure 1 further shows the token and instruction length distribution. These results demonstrate that the number of tokens in the IR file is enormous, which is much more than the statements. Besides, as the token length increases, the instruction number grows slowly. These observations motivate our code representation approach, which aims at addressing the huge file length and vocabulary size, as well as capturing the semantic relationships between code blocks.

4) The dataset is publicly available at https://zenodo.org/record/4247595#.Y74fjexBzIF.

5) The optimization level is chosen randomly per file when generating IR.

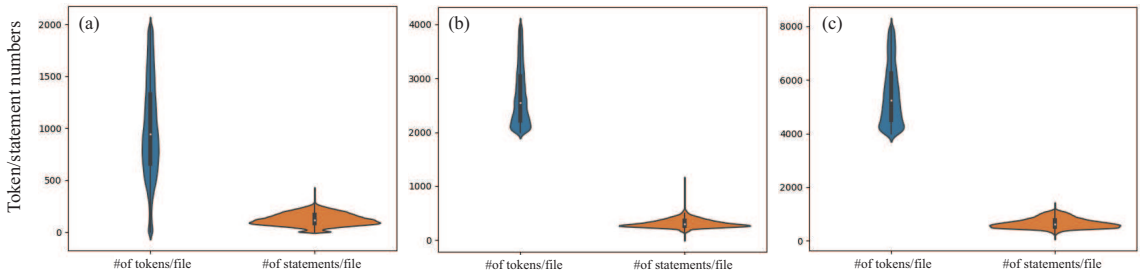


Figure 1 (Color online) Token/instruction length distribution in IR file. The data are grouped by the number of tokens for each file (0–2 k, 2–4 k, 4–8 k).

Table 2 DeepDataFlow dataset information.

	Source language	Domain	# of IR files	Cross entropy
Linux	C	Operating systems	13851	0.2271
TensorFlow	C++	Machine learning	1584	0.4971
GitHub	C/Haskell/OpenCL/Swift	Various	51222	0.332
Total	–	–	250428	0.0929

3.2 Conditional occurrences of tokens in IR instructions

To investigate where the principle of naturalness [17] also holds for IR code, we train statistical language models on the DeepDataFlow dataset and employ cross-entropy as the evaluation metric. Cross-entropy measures how well a statistical language model captures the regularities in a corpus. A good model is expected to make correct predictions on the test data with high confidence, and has a low entropy. That is, lower cross-entropy values are better. We first train a general language model on all the IR code from the whole dataset. To further analyze the breakdown performance at different domains, we train several domain-related language models on the IRs from Linux, TensorFlow, and GitHub, respectively. The cross-entropy results are shown in Table 2. Compared with the LMs trained on the source code corpus, where the cross entropy score under the comparable model settings is around 1 [9], the cross entropy score of the IR-based LMs is smaller, especially when the training data size increases. The results indicate that the IR code is mostly simple and rather repetitive, and thus the predictable statistical properties can be easier captured by statistical language models, and the learned features can further be leveraged for software engineering tasks.

4 Approach

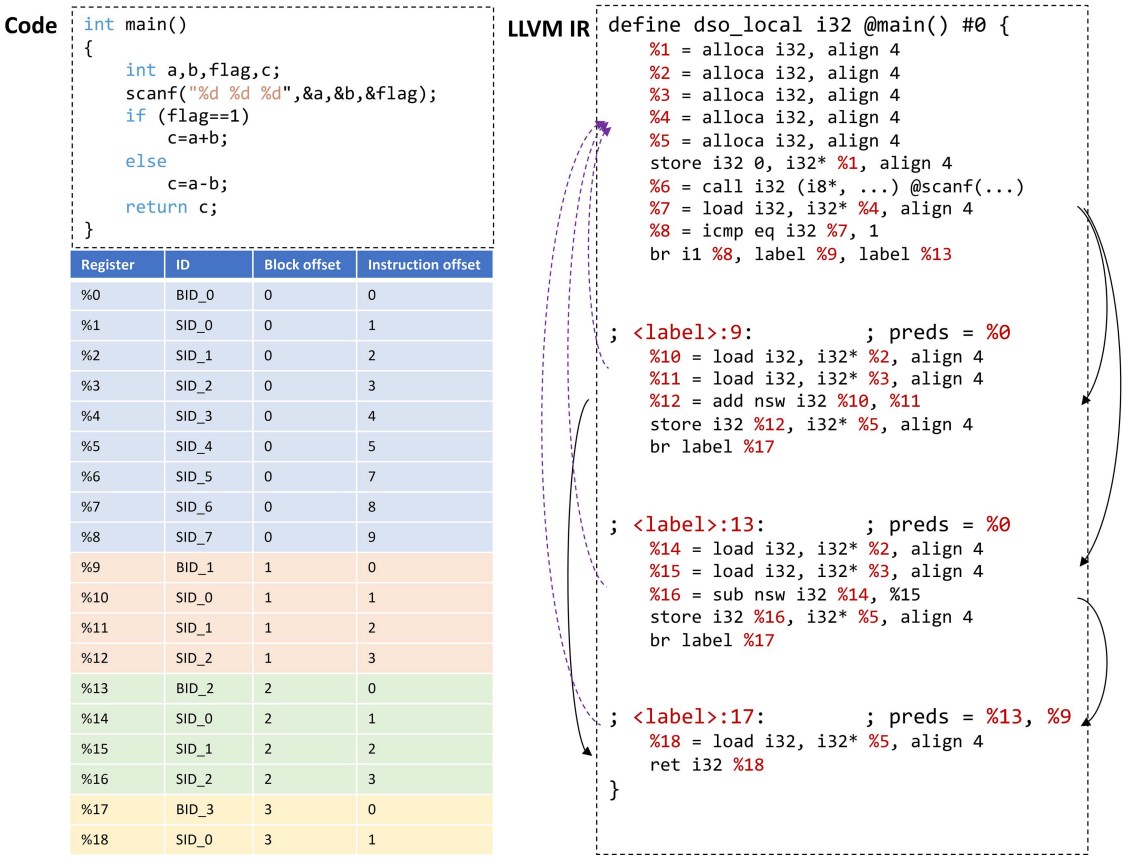
4.1 Key ideas

How to represent. For code representation, we utilize LLVM IR, which is more robust for learning code semantics compared to high-level PLs. To address the unlimited vocabulary size issue, we propose a variable identification strategy to allocate each register variable to a new ID that can represent their relative location among the file. Besides, we also extract the data dependency information among code blocks for further code semantic learning.

How to process. To learn the semantic information from the input IR data, we build a hierarchical multi-layer Transformer encoder, where a lower-level sentence-BERT encoder is responsible for producing the vector representation for each IR instruction, and an instruction-level encoder aims to capture the data dependency information as well as the code semantics through a hybrid attention mechanism.

4.2 Transforming source code into LLVM IR

We first leverage several tools (Clang and JLang) to transform the programs of both training and evaluation data into LLVM IR. When generating IRs, we employ three code generation options (O0, Oz, and Ofast) to generate IRs of different optimization levels. It is also worth noting that, LLVM IR has three different forms: as an in-memory compiler IR, as an on-disk bitcode representation, and as a human readable assembly language representation. The three different forms of LLVM are all equivalent. The



Register	ID	Block offset	Instruction offset
%0	BID_0	0	0
%1	SID_0	0	1
%2	SID_1	0	2
%3	SID_2	0	3
%4	SID_3	0	4
%5	SID_4	0	5
%6	SID_5	0	7
%7	SID_6	0	8
%8	SID_7	0	9
%9	BID_1	1	0
%10	SID_0	1	1
%11	SID_1	1	2
%12	SID_2	1	3
%13	BID_2	2	0
%14	SID_0	2	1
%15	SID_1	2	2
%16	SID_2	2	3
%17	BID_3	3	0
%18	SID_0	3	1

Figure 2 (Color online) Example of input representation in EECS.

human readable IR has been widely used in program debugging and analysis, we make use of this format for further processing.

4.3 IR-based graph construction & input representation

After generating the IR instructions, our initial step is to employ the variable identification strategy to update the names of register variables. Subsequently, we extract the data flow information and establish edges between the IR blocks that share data-flow relationships, thereby constructing the IR graph. In this graph, each node represents an IR instruction, while the edges reflect the data flow between these instructions. In the subsequent text, we will provide a detailed explanation of each step involved in this process.

LLVM IR does not use a fixed set of named register variables, instead, it uses an infinite set of temporaries named with a % character, (i.e., 1%, 2%, ...). To represent these register variables properly without using an infinite set of vocabulary, we propose a variable identification strategy, which allocates each register variable to a new ID that can represent their relative location among the file. Figure 2 shows a detailed example, where the first column lists all the register variables in the IR file, and the second column presents their corresponding IDs. Specifically, for the block label register, such as "⟨label⟩:9" and "⟨label⟩:13", we use BID_i to denote that the variable is the label of the i-th block. For the registers defined in the instruction block, such as "%1" and "%2", we use SID_i to denote that the variable is the i-th defined among the current block. Besides, we also use block offset and instruction offset embeddings to indicate the location information of each instruction, and these offset IDs are shown in the third and fourth columns in Figure 2.

To further capture the code semantics, we extract the data flow information among code blocks. Specifically, we first create edges connecting blocks that have data flow relationship according to the predecessor instruction preds = %ID at the beginning of each code block and the branch statements br label %ID at the end. The edges are denoted by the solid arrows in Figure 2. Then we connect the first block to the rest blocks, which are denoted by the dotted arrows. It is worth noting that all the connection

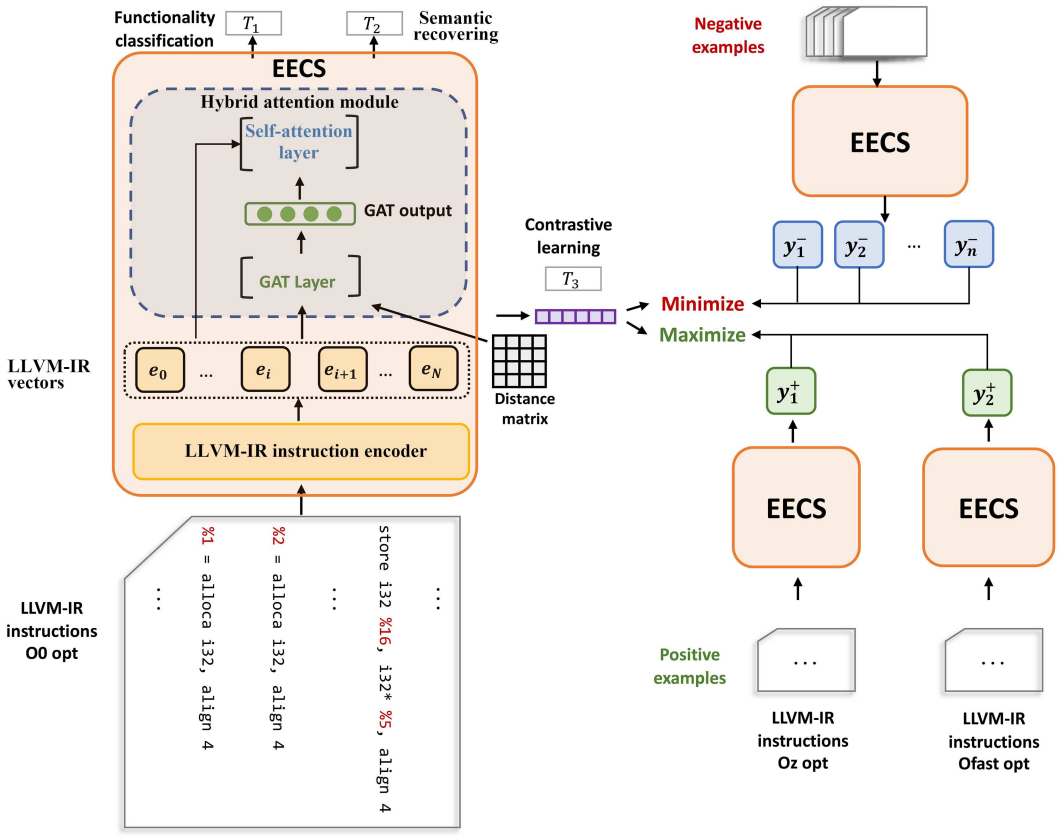


Figure 3 (Color online) Model architecture of EECS.

between blocks is performed at the first instruction of the block. In this way, the first instruction of the first code block can serve as an aggregated representation of the whole IR file.

4.4 Model architecture

The overall architecture of EECS is shown in Figure 3. EECS is built upon a hierarchical multi-layer Transformer encoder. Specifically, there are two levels of encoders, the lower level is an LLVM-IR Instruction Encoder, which is used to process tokens from IR instructions and produce the vector representation for each IR instruction, aiming at reducing the input instruction length and further reducing the time and space cost. The upper level is a hybrid attention module, which aims to capture the data dependency information as well as the code semantics from the instruction vectors via a hybrid attention mechanism.

4.4.1 *LLVM-IR instruction encoder*

As demonstrated in Section 3, the token sequence of the IR file is usually long. If we merely represent the IR code as a sequence of tokens and feed it into a neural network with sequential architecture like a Transformer encoder, the time and memory costs would be substantial. If we follow the previous work to truncate the input sequence to a fixed length, such as 512 tokens, much important information will be discarded as most of the IR files are longer than that. To handle this issue, instead of directly encoding the token sequence, we first employ a sentence-BERT [33] as the LLVM-IR instruction encoder to encode each IR instruction $x_i = \{t_1, t_2, \ldots, t_n\}$ into a vector e_i. Then, those instruction-level vectors will be fed to the hybrid attention module for further feature extraction. To make it clear, we present the time and storage costs of EECS and ProGraML, with a focus on their application in the algorithm classification task. In the context of EECS, the average number of vertices and edges per IR file is 169 and 32, respectively. These values are significantly smaller than those observed in ProGraML [26], where the average number of vertices and edges per IR file is 312 and 569, respectively. Additionally, ProGraML utilizes an extended node representation derived from inst2vec [24] with 200-dimensional embeddings. In EECS, the node representation is computed using sentence-BERT, leading to 384-dimensional embeddings. During

the evaluation process, the average time taken by EECS is 0.36 s per example, whereas ProGraML requires 0.51 s per example. Furthermore, in terms of storage costs, EECS demonstrates a reduction of approximately 20.2% compared to ProGraML.

4.4.2 *Comparison to other IR-based representations*

As a novel approach for IR-based code semantic representation learning, EECS distinguishes itself from prior related work through its unique input representation strategy and model architecture. Specifically, it stands out from previous IR-based code representations [24, 26] in the following three key areas.

• Input representation: EECS employs variable identification and data flow extraction techniques to allocate each register variable a new ID that represents its relative location. This reduces the vocabulary size of variables and allows for a more precise and cost-effective capture of code semantics. Additionally, EECS extracts data flow information among code blocks, further enhancing its ability to capture code semantics accurately.

• Input processing: to efficiently learn semantic information from IR data, EECS utilizes a hierarchical multi-layer Transformer encoder. A lower-level sentence-BERT encoder generates vector representations for each IR instruction, while an instruction-level encoder captures data dependency information and code semantics through a hybrid attention mechanism. Compared to prior IR-based code representations, EECS significantly reduces the number of vertices and edges in the IR graph, thereby lowering computational and memory costs.

• Model architecture: EECS incorporates GAT for embedding the data flow information of IR blocks. In contrast to previous approaches that utilize GGNN or RNN, GAT excels at capturing complex relationships and dependencies among graph nodes by adaptively assigning attention weights to neighbors. This capability enables EECS to effectively model long-range and non-linear dependencies between code blocks, thereby achieving superior performance in code semantic representation learning.

In summary, the combination of EECS's unique input representation strategy, efficient input processing utilizing a hierarchical Transformer encoder, and the adoption of GAT for modeling complex relationships sets it apart from prior IR-based code representations. EECS offers improved accuracy and cost-efficiency in capturing code semantics.

4.4.3 *Hybrid attention module*

Specifically, we first employ the graph attention network (GAT) [34] to embed the data flow information of the IR code blocks into a structured representation, and then use it to bias the self-attention. We build a L_G-layer GAT to compute the structured output of each instruction from e_i via a dependency-aware self-attention mechanism. Specifically, each GAT layer is responsible to learn the instruction representation by attending to their adjacent instructions:

$$O = \text{Attention}(Q, K, V, M), \tag{1}$$

where $Q = K$, the mask M is computed as

$$M_{ij} = \begin{cases} 0, & D_{ij} = 1, \\ -\infty, & D_{ij} = 0, \end{cases} \tag{2}$$

where D_{ij} indicates whether the edge exists between the i-th instruction and the j-th instruction, as we have mentioned in Subsection 4.3. $D_{ij} = 1$ indicates there is an edge between them, and $D_{ij} = 0$ means no edge exists.

Finally, we concatenate the output vectors from all the graph attention heads to produce the GAT output representations $\mathcal{G} \in \mathbb{R}^{n \times kd_g}$, where k is the number of graph attention heads, d_g is the head dimension. To maintain the lightweight nature of GAT, it solely consists of multi-head attention with no inclusion of feed-forward sub-layers or residual connections. To incorporate the structural syntax information into the model, we utilize a structure-aware self-attention approach. This method takes into account the structural relations while computing attention scores between code tokens in the following self-attention layer. Specifically, we use the representations produced by GAT to bias the self-attention:

$$O = \text{Attention}(Q + \mathcal{G}G_Q, K + \mathcal{G}G_K, V, M), \tag{3}$$

where $G_Q, G_K \in \mathbb{R}^{d_{kd_g} \times d_k}$ are the projection parameter matrices. $\mathcal{G}G_Q$ and $\mathcal{G}G_K$ are responsible for providing semantic clues to guide self-attention, where each instruction can pay more attention to instructions with semantic dependencies.

It is worth noting that we only upgrade parts of the attention heads from the backbone Transformer encoder to GAT-heads to fuse the syntax representations. All the encoder layers from the backbone Transformer are upgraded to GAT-layers, aiming to merge the syntax representations.

4.5 Training strategy

4.5.1 *Main objective: program classification*

To understand the code semantics, we train EECS with a classic program semantic distinguishing task, i.e., classifying programs by their functionalities. We employ the cross-entropy loss in this objective:

$$\mathcal{L}_{\text{CLS}} = -\sum_{i=1}^{K} y_i \log(p_i), \tag{4}$$

where K is the number of the categories, y is the ground truth, and p is the predicted results.

As the program classification task may not be sufficient to guide EECS to learn the code semantic knowledge, we also adopt the following two auxiliary objectives that enhance the model to capture the code semantic information.

4.5.2 *Contrastive learning objective*

When using the LLVM compiler to generate IR, different optimization levels (O0, O1, O2, O3, Ofast, Oz, Os) can be applied, aiming to optimize the generated IR code by reducing code size or running time. For example, O0 means no optimization, this level compiles the fastest and generates the most debuggable code. O2 is a moderate level of optimization that enables most optimizations. Oz is like O2, but reduces code size further. These IRs generated with different optimization levels are naturally positive samples, whose semantics are exactly the same, but with different formats. We argue that these IRs are semantically equivalent, and should have similar underlying representations. Contrastive learning can induce semantic invariance into the model by minimizing the distance between the representations of similar examples while maximizing the distance between dissimilar examples [35]. As shown in Figure 3, we use the O0 optimized IR as the query code, Oz and Ofast as the positive examples, and adopt in-batch negatives to calculate the InfoNCE contrastive loss [36]:

$$\mathcal{L}_{\text{CT}}(q, c^+, c_1^-, c_2^-, \ldots, c_n^-) = -\log \frac{\exp(q \cdot c^+/t)}{\exp(q \cdot c^+/t) + \sum_{c^-} \exp(q \cdot c^-/t)}, \tag{5}$$

where $q = f(x_{\text{O0}})$ is the query representation, which is computed by EECS network f, x_{O0} is the query code. $c^+ = f(x_{\text{Oz}} || x_{\text{Ofast}})$ is the positive code representation, and we randomly choose between the Oz or Ofast IR to compute c^+. c_i^- is computed using other inputs in the mini-batch.

4.5.3 *Code SR objective*

When converting the source code to IR, the variables and identifiers are recoded as meaningless temple register identifiers, for example, %1, %2, etc. The semantic information implied in those variables is lost after the converting process. After the register reallocation in Subsection 4.3, these registers are replaced by the location ID. The semantic information is still not included. To compensate for the information loss, we propose the code SR objective, which aims to recover the original source code according to the IR representations, and further enhances the model to learn sufficient semantic knowledge from the IR. Specifically, we initialize a multi-layer Transformer decoder to auto-regressively generate the source code tokens based on our EECS output and previously generated tokens. The loss is computed as

$$\mathcal{L}_{\text{SR}} = \sum_{t=1}^{T} \log p(y_t | y_{<t}, c; \theta). \tag{6}$$

The parameters of EECS are learned to minimize the sum of the losses of the above three objectives, and are shared among all the tasks. The final loss function is computed as follows:

$$\mathcal{L} = \mathcal{L}_{\text{CLS}} + \mathcal{L}_{\text{CT}} + \mathcal{L}_{\text{SR}}. \tag{7}$$

Table 3 Optimization level in LLVM IR generation of CLang.

Optimation level	Meaning
-O0	Means "no optimization": this level compiles the fastest and generates the most debuggable code.
-O, -O1	Somewhere between -O0 and -O2.
-O2	Moderate level of optimization which enables most optimizations.
-O3, -O4	Like -O2, except that it enables optimizations that take longer to perform or that may generate larger code (in an attempt to make the program run faster).
-Ofast	Enables all the optimizations from -O3 along with other aggressive optimizations that may violate strict compliance with language standards.
-Os	Like -O2 with extra optimizations to reduce code size.
-Oz	Like -Os (and thus -O2), but reduces code size further.
-Og	Like -O1. In future versions, this option might disable different optimizations in order to improve debuggability.

5 Experimental setup

5.1 Model configuration

We employ sentence-BERT [33] of the "all-MiniLM-L6-v2" version in the IR instruction encoder. For the hybrid attention module, we set the number of both the self-attention layer and the GAT layer to 8. The number of decoder layers in the SR objective is also set to 8. The number of self-attention heads and GAT heads is set to 8 and 1, respectively. The embedding size is 384. We optimize EECS with the Adam optimizer [37] with the initial learning rate of 5e−5. For the SR task, we initialize an 8-layer Transformer decoder to generate the source tokens. Our experiments are implemented on 2 Nvidia V100 GPUs with Pytorch. We adopted early stop strategy, where the training was terminated if the validation accuracy did not improve for 3 consecutive epochs. Finally, EECS was trained for 18 epochs. After training was completed we selected the checkpoint with the highest accuracy on the validation set to use for testing or further fine-tuning.

5.2 Data pre-processing

For C++ code in the POJ dataset, we use LLVM Clang[6)] to emit LLVM IR. For Java code in the GeeksforGeeks dataset, we use Jlang[7)] and Polyglot[8)] to translate Java into LLVM IR. When using the LLVM compiler to generate IR, different optimization levels can be applied, aiming to optimize the generated IR code by reducing code size or running time. Table 3 shows the optimization level options and their meanings of CLang[9)]. We employ three code generation options (O0, Oz, and Ofast) to generate IRs of different optimization levels. The max input instruction length is set to 512. After getting the IR instructions, we first apply the variable identification strategy as described in Subsection 4.3 to update the register variables' name. Then we extract the data flow information and add edges between the IR blocks that have data-flow relationships to build the IR graph, where the node represents an IR instruction, and the edge represents the data-flow between the instructions.

5.3 Tasks and setup

5.3.1 *Algorithm classification*

This task refers to the classification of source code into a predefined number of classes based on the task it solves. In this task, we use the POJ-104 dataset [5], which contains the C/C++ source code collected from 104 programming problems in the open judge system[10)]. The target label of each program is their problem ID (0–103), and the programs from the same problem have the same functionality. There are 500 programs in each problem, and 52000 programs in total. The programs are randomly split by 8:1:1 for training, validation, and testing. To convert the programs to the LLVM IR, we compile the source code using Clang[11)] with -O0, -Oz, -Ofast optimization flags to generate the corresponding IRs.

6) https://clang.llvm.org/.
7) https://polyglot-compiler.github.io/JLang/.
8) https://www.cs.cornell.edu/projects/polyglot/.
9) A C language family frontend for LLVM.
10) http://poj.org/.
11) https://clang.llvm.org/.

As mentioned in Subsection 4.3, the representation of the first instruction can be viewed as the aggregation of each program. Thus, we use the sum of the first instruction representations from the three IR inputs (O0, Oz, and Ofast) to make predictions about the problem ID. We compare EECS with 5 advanced unpretrained program representation approaches, including tree-based convolutional neural network (TBCNN) [5], NCC [24], Tree-Transformer [38], and ProGraML [26]. Cummins et al. [26] also found that lifting the XFG graph representation from NCC [24] to the task of algorithm classification showed a strong improvement. Thus, we also included XFG as another baseline. These models have been evaluated in the POJ-104 dataset in their original papers, and the results of these baseline methods are directly cited from Cummins et al. [26] or their original papers. Besides, we also choose 3 powerful pre-trained models: $\text{CodeBERT}_{\text{base}}$ [13], PLBART [15], and $\text{CodeT5}_{\text{base}}$ [14]. Comparing our approach to other large language models (LLMs), such as Codex [39] and CodeGen [40], presents a challenge due to their pre-training on a vast collection of source code from GitHub. This poses a risk of data leakage, as it could potentially overlap with our evaluation set. Therefore, ensuring a fair and unbiased comparison becomes difficult. Although the model sizes and pre-training corpora of our selected models (CodeBERT [13], PLBART [15], and CodeT5 [14]) are larger than those in EECS, it is important to note that our evaluation data corpus is not included in their pre-training data. This distinction is significant, as it supports a fair comparison. As a result, we have chosen these models as our baselines for comparison. We fine-tune these models on POJ-104 source code training set[12]. Our model is trained on the POJ-104 training IR dataset, we directly run our trained model on the POJ test IR dataset in this experiment. Following ProGraML [26], we employ the error rate to measure the classification performance. Besides, we also present the relative improvement compared with the results of ProGraML, which achieves previous state-of-the-art performance on this task.

5.3.2 *Heterogeneous compute device mapping*

Given an OpenCL kernel program, the objective of the heterogeneous device mapping (DEVMAP) is to decide whether the given program will run faster on the CPU or GPU. This problem can be formed as a binary classification task. We use the OpenCL dataset [41] in this experiment, which consists of labeled CPU/GPU instances for 256 OpenCL kernels from AMD and NVIDIA[13]. Each dataset consists of 680 labeled instances that are derived from the 256 unique kernels by varying dynamic inputs. Each program in the dataset has one corresponding IR code. In this experiment, we do not use extra IR formats. Compared with algorithm classification, DEVMAP is more challenging in both its difficulty and the corpus size (680 examples for each dataset).

We use the representations of the first instruction to make predictions about the running device. We compare EECS with 3 advanced program representation approaches: DeepTune [41], NCC [24], and ProGraML [26]. Besides, we also include $\text{DeepTune}_{\text{IR}}$, which adapts the DeepTune model using the tokenized LLVM-IR as inputs. These models have been evaluated in the DEVMAP dataset in their original papers, and the results of these baseline methods can be directly cited from Cummins et al. [26]. Besides, we also choose 3 pre-trained models: CodeBERT [13], PLBART [15], and CodeT5 [14], and fine-tune them on the source code of the DEVMAP training set. In this experiment, we reused our previous well-trained model (trained on the POJ dataset). Then we fine-tune it on the IR instructions of the DEVMAP training set, and evaluated it on the test set. To measure the effectiveness of the models, we adopt accuracy as the metric, which computes the percentage of correct device predictions.

5.3.3 *Program translation*

This task refers to translating functions from one PL to another. In this paper, we focus on the Java to C++ translation and use the GeeksforGeeks dataset provided by Rozière et al. [32]. There are 627 Java-C++ source code file pairs in the dataset, where 4 Java files are uncompilable. Finally, we have 623 pairs. The programs are split by 8:1:1 for training, validation, and testing. To convert the Java programs to LLVM IR, we compile the source code using JLang with the default optimization level. This task differs from the previous two tasks in both the language of the source code (C family vs. Java) and the

12) Since these models are originally pre-trained on the source code tokens instead of IR instructions, to keep the input consistent, we fine-tune them on the source code tokens.

13) AMD set uses an Intel Core i7-3820 CPU and AMD Tahiti 7970 GPU, and NVIDIA set uses an Intel Core i7-3820 CPU and an NVIDIA GTX 970 GPU.

Table 4 Results on the algorithm classification task, including the error rate and the relative improvement compared with ProGraML. The numbers in bold indicate the best performance.

Model	Error rate (%)	Improvement (%)
TBCNN	6.0	–
NCC	5.2	–
XFG	4.3	–
Tree-Transformer	3.9	–
ProGraML	3.4	0.0
$\text{CodeBERT}_{\text{base}}$	3.0	11.8
PLBART	2.9	14.7
$\text{CodeT5}_{\text{base}}$	2.7	20.6
EECS	**2.4**	**29.4**
- RI	2.9	14.7
- GAT	3.1	8.8
- CT	3.0	11.8
- SR	2.9	14.7

target (predict a class label vs. generate a complete function), which can be used to assess the semantic preserving and generalization capability of EECS.

The experimental setting of this task follows the setting in the code SR objective, where the Transformer decoder is reused for generating the target C++ functions. We compare EECS with 3 advanced approaches: TransCoder [32], CodeT5 [14], and PLBART [15]. TransCoder is an unsupervised program translation approach, that leverages huge monolingual programs corpus pre-train their model, covering C++, Java, and Python languages. Then they employ back-translation to tune the model. It has achieved advanced performance in function translation between C++, Java, and Python. TransCoder has been evaluated in the whole GeeksforGeeks dataset in their original paper. To keep the results comparable with other baselines, we re-run their model with greedy and beam search decoding strategy on our re-split GeeksforGeeks test set. PLBART [15], and CodeT5 are fine-tuned on the source code pairs of the training set. In this experiment, we reused our previous well-trained model (trained on the POJ dataset). Then we fine-tune it on the IR instructions of the GeeksforGeeks training set and evaluated it on the test set. To measure the effectiveness of the models, we adopt BLEU score, CodeBLEU score, and exact match accuracy (EM Acc) as the metrics.

6 Experimental results

6.1 RQ1: effectiveness and practicality in semantic understanding

To verify the effectiveness and practicality of EECS in code semantic understanding, we apply our approach to two typical code semantic understanding tasks, algorithm classification and heterogeneous computing device mapping, and compare it with advanced code representation approaches. The experimental results of algorithm classification are shown in Table 4. Among all the unpretrained baselines, ProGraML performs best. When applying existing pre-trained models, all of these models achieve better performance than ProGraML, where CodeT5 outperforms ProGraML by a large margin. The results of these pre-trained models demonstrate the effectiveness of the pre-training techniques. Nonetheless, EECS can outperform the above baselines, including the powerful pre-trained models. Our model is trained on the POJ training dataset from scratch, which is much smaller than the pre-training corpus used in those pre-trained models. Nevertheless, the efficient code semantic representation approach and several auxiliary training objectives help EECS extract sufficient code semantic knowledge from the limited training corpus, thus our model can outperform those baseline models substantially.

To further assess the performance and practicality of EECS, we then evaluate our model on a more challenging heterogeneous computing device mapping task. The results of heterogeneous computing device mapping are shown in Table 5. For the AMD set, ProGraML achieves the best performance, and our model obtains comparable results. For the NVIDIA set, our model outperforms the baseline by a large margin. In this experiment, the performance of all the pre-trained models is worse because the number of training programs is not big enough to tune the pre-trained models well. Despite the difficulties, EECS can still perform well, which further demonstrates EECS has great practicality and

Table 5 Results on heterogeneous compute device mapping. The numbers in bold indicate the best performance.

Model	AMD Acc (%)	NVIDIA Acc (%)
DeepTune	71.9	61.0
$\text{DeepTune}_{\text{IR}}$	73.8	68.4
NCC	80.3	78.5
ProGraML	**86.6**	80.0
$\text{CodeBERT}_{\text{base}}$	61.0	72.8
PLBART	54.4	76.5
$\text{CodeT5}_{\text{base}}$	58.1	78.7
EECS	85.3	**85.3**
- RI	83.4	84.8
- GAT	83.0	84.5
- CT	82.4	83.3
- SR	84.7	81.0

Table 6 Results of Java → C++ program translation.

	BLEU (%)	EM Acc (%)	CodeBLEU (%)
TransCoder (greedy)	84.2	24.2	70.7
TransCoder (beam 5)	87.6	25.8	73.5
CodeT5	72.8	11.3	65.5
PLBART	71.3	9.7	64.3
EECS	73.5	12.9	69.2
- RI	71.6	9.7	68.9
- GAT	71.8	11.2	67.0
- CT	70.3	11.2	67.1
- SR	69.6	8.1	66.2

can perform well in the presence of data scarcity.

6.2 RQ2: generalization capability analysis

EECS is trained on the IRs generated by the C++ language. In RQ1, EECS is also evaluated on the C language family, i.e., C++ in algorithm classification and OpenCL in DEVMAP. To figure out how well our approach performs in unseen language scenarios, in this RQ, we evaluate the generalization capability of EECS through a new experiment on the Java language. Specifically, we choose Java to C++ function translation task. The results are shown in Table 6.

Compared with the two classification tasks in RQ1, the translation task is more challenging, reflected in both the new source language and the longer generation objective. All of the baselines are pretrained with a large corpus, where TransCoder uses cross-language pretraining objectives and a back-translation augmentation, thus achieving the best performance in all the metrics. CodeT5 and PLBART have comparable performance, and both are worse than our model, especially in terms of CodeBLEU score. Although EECS has never seen any IR instructions for the Java programs during pre-training, the underlying semantics of the LLVM IR instructions for different languages are similar. Thus the knowledge acquired from the IRs (converted from C++) can be transferred to learn the new IRs (converted from Java). Besides, the SR task can further teach EECS how to convert the IR representations to the C++ source code, which can benefit the translation generation process. Thus, our model demonstrates a similar performance to TransCoder in terms of CodeBLEU. Specifically, it achieves an 88.6 Semantic Data-flow Match score, while TransCoder achieves a slightly higher score of 93.9. The CodeBLEU score is a more reliable metric for assessing semantic correctness compared to traditional scores like BLEU and EM ACC. The results demonstrate that extracting and modeling the semantic feature from LLVM IRs is effective in enhancing the cross-lingual transfer ability of the code representation models, leading to great performance in unseen source language understanding scenarios, and further demonstrates EECS has great generalization capability.

Table 7 Algorithm classification accuracy for varying IR lengths. The numbers in bold indicate the best performance.

Length	EECS (%)	$\text{CodeT5}_{\text{base}}$ (%)
0–100	91.24	**91.54**
100–200	97.00	**97.41**
200–300	**98.19**	98.15
300–400	**97.93**	97.66
400–500	**96.86**	96.41
500–600	93.65	93.65
600+	**96.85**	93.75

Table 8 Accuracy of the device mapping for varying IR lengths.

	EECS (%)		CodeT5 (%)	
	AMD	NVIDIA	AMD	NVIDIA
0–100	85.7	90.5	83.3	88.1
100–200	80.0	67.5	75.0	65.0
200–300	83.3	95.8	79.2	91.7
300–400	44.4	77.8	44.4	77.8
400–500	100.0	100.0	100.0	100.0
500–600	100.0	100.0	100.0	100.0
600+	81.3	87.5	75.0	81.3

6.3 RQ3: ablation study

To evaluate the contribution of the proposed components in EECS, i.e., register identification (RI) strategy in input representation, GAT in hybrid attention module, contrastive learning (CT) training objective, and SR objective. We conduct an ablation study by removing each component from EECS during training. The results of two tasks are shown in the last four rows of Tables 4–6. As seen from the results, removing each component leads to a performance drop, demonstrating that all these modules are effective. Specifically, for the algorithm classification task, GAT and CT bring the most improvement. The results suggest that the data flow information between the IR code blocks and distinguishing the similar or dissimilar IR codes are essential for understanding the functionality of the program. For the device mapping task, CT and SR contribute more to the performance improvement. For the program translation task, SR contributes the most. The ablation results of these two tasks further show that the SR objective plays an important role in the difficult downstream tasks, where the code semantic information is hard to understand.

6.4 RQ4: sensitivity analysis

In this RQ, we analyze how various factors affect EECS's performance, including the IR lengths and training data size.

6.4.1 *Performance of different IR lengths*

Recall that from Section 3, the number of IR instructions is typically enormous, how to encode IR instruction efficiently and effectively is an important objective in EECS. In this subsection, we take a closer look at the performance of different IR lengths, i.e., the number of IR instructions. Due to the small scale of the test set for the program translation task, and the small difference in IR length, we only conduct experiments in algorithm classification and device mapping tasks in this section. Tables 7 and 8 present the performance of EECS and CodeT5 of varying lengths for algorithm classification and device mapping tasks, respectively. For algorithm classification, the performance fluctuates of EECS are less than CodeT5 as IR length changes, and the improvements over CodeT5 become bigger as the length increases. We can observe similar results on the device mapping task. These results demonstrate that through the effective representation strategy, EECS can maintain a considerate performance as the input length grows.

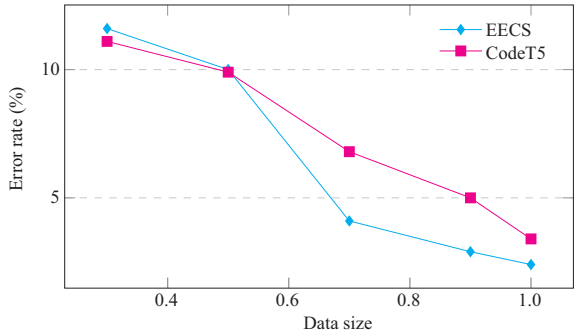


Figure 4 (Color online) Algorithm classification results of different training data sizes.

Table 9 Results of different model sizes and architectures in EECS. The numbers in bold indicate the best performance.

	Model	Error rate (%)
Impact of model size	EECS (En8-De8)	2.4
	En6-De6	2.8
	En12-De12	**2.2**
Impact of model architecture	EECS (GAT)	**2.4**
	GAT $\rightarrow$ GGNN	2.6
	GAT $\rightarrow$ GCN	3.0

6.4.2 *Effect of training data size*

To test the effect of the training data size, we conduct an experiment of training our model using different proportions of the training data, and present the algorithm classification results of corresponding settings in Figure 4. We also present the results of CodeT5, where the same data is used during the fine-tuning. As seen from the results, when using 30% of the original training data, the performance of CodeT5 is better. The reason might lie in that the data is not big enough to train EECS well since it is trained from scratch. However, CodeT5 can perform better by utilizing the prior knowledge obtained during pre-training. As the size of the training data increases, the error rate of both the two models drops, and EECS surpasses CodeT5 when more than 50% of the training data are used. The results demonstrate that by encoding the semantic information into the model, EECS can grasp sufficient semantic knowledge using relatively small-scale training data.

6.4.3 *Impact of model architecture and size*

In order to explore the influence of model size and architecture on performance, we have conducted a series of experiments. Initially, we varied the encoder and decoder layers of EECS to obtain insights into their impact. Additionally, we conducted another experiment to investigate the effects of different architectures on performance by replacing the GAT of EECS with alternative models, such as GGNN [42] and GCN [43]. The results are presented in Table 9.

The results obtained from the experiments reveal interesting findings regarding the impact of model sizes and architectures on performance. When investigating the effect of model sizes, it was observed that reducing the number of encoder and decoder layers from 8 to 6 led to an increase in the error rate from 2.4% to 2.8%. On the other hand, increasing the layers to 12 resulted in improved performance, achieving the lowest error rate of 2.2%. These results indicate that larger model sizes can contribute to better performance in thus task, suggesting that deeper networks are capable of capturing more complex patterns and representations. In terms of model architecture, replacing the GAT with an alternative GNN model led to a slight increase in the error rate to 2.6%. Furthermore, using GCN as a replacement for GAT resulted in a higher error rate of 3.0%. These findings demonstrate that the GAT architecture, as utilized in our model, is the most effective among the tested alternatives. This suggests that the attention mechanism employed in GAT plays a crucial role in capturing relevant information and achieving better performance in our task.

To summarize, the conducted experiments highlight the influence of both model sizes and architectures on performance. Larger model sizes demonstrate improved performance, albeit with higher memory and time costs. Moreover, the GAT architecture outperforms alternative models like GGNN and GCN,

Table 10 Comparison results with scalable GNNs on the algorithm classification task. The number in bold indicates the best performance.

Model	Error rate (%)
ProGraML	3.4
GCN	6.1
LightGCN	6.6
EECS	**2.4**

showcasing the significance of the attention mechanism in capturing relevant information. These insights provide valuable guidance for optimizing the model and selecting appropriate architectural choices for future applications.

7 Discussion

7.1 Comparison with scalable GNNs

A line of work achieves large-scale GNNs by designing scalable GNN models. They use simplification [43–45], quantization [46, 47], sampling [48, 49], and distillation [50, 51] to design efficient models. Among these models, the graph convolutional network (GCN) [43] has emerged as a popular formulation of GNNs and is being widely utilized in various tasks. Building upon GCN, LightGCN [44] simplifies the design, making it more concise and appropriate. In order to evaluate the effectiveness of existing scalable GNNs in addressing the challenges of IR-based code semantic representation, we conduct experiments by applying both GCN and LightGCN to the algorithm classification task. The graph is built based on ProGraML [26], where the vertices include statements, identifiers, and immediate values, and the relations between vertices are edges. We then compare the performance of these models with our proposed approach. The results are shown in Table 10. From the results, we notice that both our model and ProGraML outperform two scalable GNN models by a large margin. By simplifying the computation process, the two efficient GNNs lose their performance compared with ProGraML, demonstrating that the complex computation process in GNNs, as in ProGraML, is crucial for achieving good performance in the code semantic representation task. Our model, on the other hand, by leveraging an efficient input representation strategy and multi-task contrastive learning objectives, is able to capture more fine-grained semantic information and achieve higher accuracy in algorithm classification.

7.2 Training cost analysis

In our study, we have utilized a multi-task learning strategy to train our model on three distinct tasks. It is important to note that this approach introduces an increase in training time compared to training on a single task. To provide a comprehensive analysis of the additional training cost associated with multi-task learning, we present the training time of our model, as well as the time taken to train when removing several tasks. Additionally, we present the training (fine-tuning on algorithm classification task) time of two baseline models, namely CodeT5 and PLBART. It is worth noting the training times for the other baseline models were either not available in their original papers, or running with different settings and hardware configurations. For this reason, we only report the fine-tuning time of CodeT5 and PLBART, which were trained on the same devices. Both the training of our model and the fine-tuning of CodeT5 and PLBART are performed on the algorithm classification task. The models in this experiment were trained (fine-tuned) on 2 V-100 GPUs. We applied an early stopping criterion, where training was halted if the accuracy of the validation set did not improve for 3 consecutive epochs. Table 11 presents the training (fine-tuning) time of different models, along with the number of training epochs.

As seen from the results, the number of training epochs varies across different model settings. While CodeT5 and PLBART exhibit similar model sizes, PLBART achieves convergence more rapidly, resulting in a reduced fine-tuning time requirement. In contrast, our model boasts a smaller model size compared to CodeT5 and PLBART. Although it necessitates more epochs to converge, the total training time is not significantly increased and, in fact, is even faster than fine-tuning CodeT5. Notably, when we removed the CT and/or SR tasks from EECS, there was a slight decrease in training time, accompanied by a corresponding decline in performance. This emphasizes the need to carefully consider the trade-off between training time and the benefits derived from multi-task learning. Despite the additional cost in

Table 11 Training statistics of different models.

Model	Error rate (%)	Training (fine-tuning) time (h)	Training epoch
CodeT5	2.7	4.6	16
PLBART	2.9	3.9	13
EECS	2.4	4.3	18
- CT	3.0	2.9	16
- SR	2.9	3.4	18
- CT&SR	3.1	2.6	15

terms of training time, the enhanced performance and knowledge-sharing capabilities of our multi-task model affirm its applicability in various scenarios.

8 Threats to validity

Threats to external validity relate to the generalizability of our approach and results. To mitigate this threat, we evaluate EECS on three different tasks, covering three source languages (C++, OpenCL, and Java). The evaluation results demonstrate the effectiveness and generalizability of our model. However, the source code used in our approach must be compilable, and the PLs should have LLVM compilers. Many commonly used PLs can be compiled by LLVM compilers, such as C/C++, OpenCL, C#, Rust, Java, and Ruby. For those languages that do not currently support compilation to LLVM-IR, we believe that the LLVM compilers for them will be developed in the near future, aiming to benefit from LLVM's superior characteristics. Thus, there is a minor threat to the generalizability of EECS.

Threats to internal validity include the influence of the model hyper-parameter settings and the model architecture designing choices. We decide the hyper-parameters and architecture via small-range random grid search and manual selection. Thus, there might be little threat to the hyper-parameter choosing, and there could be room for further improvement. Besides, there also exist two concerns including the larger size of IR instructions compared to source code and the prevalence of incomplete code fragments in practical scenarios. Unlike source code, IR instructions tend to have a larger number of instructions due to the lower-level nature of the representation. The longer sequences of IR instructions may require more complex modeling techniques to effectively capture the dependencies between them. To mitigate this threat, EECS employs an efficient input representation and learning approach which reduces the input length and the vocabulary size significantly via a variable identification strategy and GAT-based multi-layer encoder. Furthermore, in practical scenarios, it is not uncommon to encounter incomplete or partially available code fragments, making it challenging to obtain accurate LLVM IRs. This could potentially affect the performance and accuracy of our approach, as incompleteness in code fragments may lead to incomplete or inaccurate LLVM IR representation. Further investigation and refinement are necessary to fully understand and overcome these challenges for more reliable and effective code representation.

Threats to construct validity relate to the suitability and effectiveness of the evaluation measure. To make a fair comparison with the advanced baselines, we employ the measures that are generally used in the previous related work for all three tasks. Specifically, we use the error rate and accuracy in the algorithm classification and device mapping tasks following [26, 31]. For program translation, we use the BLEU-4 score and exact match accuracy following [14, 32]. Another threat relates to the selection of baselines. In this paper, we compare EECS with several advanced pre-trained CLMs, where larger-scale language models like OpenAI's GPT models are not included. These models have shown amazing performance in many code-related tasks under zero-shot settings recently. The reason lies in that these models are trained in a generative manner, which is not suit for the classification tasks in our experiments. Besides, these models are pre-trained using all source code in GitHub, and keep updating with the latest open-source data and user feedback. Thus, the programs in the test set of our tasks could have been seen during training. Nonetheless, the exclusion of larger-scale language models from evaluation may limit the scope and accuracy of the evaluation results. Future evaluations may need to incorporate these models to provide a more comprehensive assessment of code semantic understanding performance.

9 Related work

Code representation learning is essential for applying machine learning techniques to various software engineering tasks. Inspired by the success of deep learning in the NLP field and the growing availability of open-source code, modeling source code with deep-learning techniques has attracted increasing attention. Prior studies mainly focus on learning representations from superficial PL features, such as source code (sub-)tokens. Iyer et al. [52] proposed to represent the token sequence by LSTM, and then evaluated the representation in the code summary generation task. Allamanis et al. [10] designed a CNN with an attention mechanism to capture the hierarchical structure of code over the subtokens. These approaches tokenize the program into sequential tokens using lexical tokenizers. Later, there is a growing body of work trying to incorporate code syntax structure into DNN through modeling ASTs [18, 19] or Graphs [22, 23]. Mou et al. [5] proposed a TBCNN to represent the ASTs of code and applied the learned representation for the algorithm classification task. Alon et al. [53] employed AST paths that are extracted by sampling on the trees to represent the programs, and then used word2vec to learn the semantics of the paths. Furthermore, Peng et al. [54] proposed TPTrans to capture the structural information of source code by integrating path encoding in Transformer. Ma et al. [55] introduced GraphCode2Vec, a self-supervised pre-training approach that produces task-agnostic embeddings incorporating both lexical and program dependence features. This achievement is made possible through the synergistic combination of code analysis and GNNs. Zhang et al. [56] proposed a novel method to represent programs using a heterogeneous program graph (HPG), which explicitly captures node and edge types. They employed a heterogeneous graph Transformer architecture to generate representations based on HPG, enhancing the expressiveness of program representations. Most recently, Wang et al. [38] introduced Tree-Transformer, a novel recursive tree-structured neural network for learning vector representations of source codes. Their approach surpasses existing tree-based and graph-based program representation learning methods in both tree-level and node-level prediction tasks, demonstrating its superior performance. Cheng et al. [57] conducted a comprehensive exploration of the disparity between conventional tools and existing learning-based code representation approaches in the field of vulnerability detection. Through their empirical study, they identified several crucial issues and challenges associated with developing classification models that accurately identify bug-triggering paths. These findings act as a significant call to action, urging researchers and practitioners to dedicate efforts towards developing more sophisticated learning-based methods specifically tailored for domain-specific tasks.

Inspired by the success of pre-trained language models in NLP, many recent code representation learning works employ pre-training on large-scale code corpus and achieve promising results in many code-related tasks. Feng et al. [13] proposed CodeBERT, which is a bimodal pre-trained model for both PL and NL. Later, Guo et al. [27] proposed GraphCodeBERT, aiming at capturing the code semantics better by incorporating the data-flow information into the pre-training model. CugLM [58] employs a shared Transformer for both understanding and generating objectives by adjusting the attention mask matrix, and also proposed an IP objective to learn the semantic information. CodeT5 [14], which is built on T5 [12], is a unified pre-trained encoder-decoder model that can support both code understanding and generation tasks. The model is trained with an identifier-aware objective that considers the token type information, and also utilizes the NL-PL bimodal dual generation task to learn the NL-PL alignment. It has achieved state-of-the-art performance on many tasks. Wan et al. [59] conducted a thorough structural analysis aiming to provide an interpretation of the above code pre-trained language models, try to figure out why these models work and what feature correlations they can capture.

To represent code semantics in a robust manner, there is a broad line of work attempting to understand code based on IRs. VenkataKeerthy et al. [31] proposed an LLVM-IR based framework IR2VEC, which incorporates the data-flow analysis results into the code representation learning process, aiming at capturing the syntax and the semantics of the programs. Sui et al. [25] presented Flow2Vec, which learns value-flow embedding via matrix multiplication, aiming at preserving context-sensitive transitivity through CFL reachability on the interprocedural value-flow graph (IVFG). The IVFG is built on top of the LLVM IR using Andersen's pointer analysis [60]. ProGraML [26] is a language-agnostic graph representation based on LLVM IR. Different from previous approaches, it could mimic the behavior of typical iterative data-flow analyses by allowing the propagation of information through code graphs. Thus, it can accurately capture the semantics of a program's statements and the relations between them. ProGraML has achieved state-of-the-art performance in heterogeneous device mapping and algorithm classification tasks. Cheng et al. [61] introduced a novel approach for code representation called ContraFlow. This

approach involves selecting and preserving feasible value-flow paths using a pretrained path embedding model and self-supervised contrastive learning. By doing so, they effectively reduce the reliance on labeled data for training downstream models, particularly for path-based vulnerability detection.

10 Conclusion and future work

In this paper, we propose an efficient and effective code semantic learning approach based on the LLVM IR and a hybrid attention mechanism. For input representation, we propose variable identification and dependency extraction methods, aiming at extracting the code semantics effectively from the IR instructions. Then we build a hierarchical multi-layer Transformer model to capture the data dependency information as well as the code semantics through a hybrid attention mechanism. During the training, we employ three training objectives to enable the model to learn code semantics and functionality more robustly. Experimental results on three tasks demonstrate the remarkable capability of EECS in program semantics understanding.

There is still room for enhancing the strength of EECS. In future work, we will design more effective mechanisms to further improve the accuracy of code semantic understanding by incorporating program execution information and description of the code function (e.g., comments), which are important for in-depth code semantic understanding and can benefit many SE tasks. Besides, we also plan to extend our approach to broader applications, for example, malicious code detection, vulnerability detection, etc.

Recent advancements in LLMs [39, 40, 62] have gained attention in both NLP and code-related tasks. These advancements also offer implications for enhancing our proposed code representation learning approach. On the one hand, LLMs pretrained on large-scale language tasks can be fine-tuned on code-specific tasks using LLVM-IR representations. This transfer-learning approach allows the model to capture the code-specific semantics of LLVM-IR while leveraging the language understanding capabilities of LLMs. On the other hand, integrating LLMs into LLVM-IR based code representation learning can enhance the interpretability and generation of LLVM-IR code by leveraging their language understanding capabilities. This can lead to more human-readable and contextually-aware representations that improve code comprehension. Moreover, LLMs also have showcased success in multimodal architectures that combine code and NL. Future work can also investigate cross-modal learning techniques to integrate our LLVM-IR representations into these multimodal frameworks. In this way, the model can learn more comprehensive code representations that capture both the high-level semantics of LLVM-IR and the context from other modalities. Overall, recent progress in LLMs presents opportunities for advancing the quality, comprehensiveness, and contextual awareness of LLVM-IR based code representations. All the datasets and implementation code are available at figshare link: https://figshare.com/s/822417778045fa65a3eb.

Acknowledgements This work was supported by National Natural Science Foundation of China (Grant Nos. 62302021, 62177003, 62072007, 62192733, 61832009, 62192731, 62192730).

References

1 Wang W H, Li G, Ma B, et al. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In: Proceedings of the IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2020. 261–271
2 Yu D J, Yang Q X, Chen X, et al. Graph-based code semantics learning for efficient semantic code clone detection. Inf Softw Tech, 2023, 156: 107130
3 Ahmad W, Chakraborty S, Ray B, et al. A transformer-based approach for source code summarization. In: Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, 2020. 4998–5007
4 Son J, Hahn J, Seo H, et al. Boosting code summarization by embedding code structures. In: Proceedings of the 29th International Conference on Computational Linguistics, 2022. 5966–5977
5 Mou L L, Li G, Zhang L, et al. Convolutional neural networks over tree structures for programming language processing. In: Proceedings of the 30th AAAI conference on artificial intelligence, 2016
6 Wang D Z, Jia Z Y, Li S S, et al. Bridging pre-trained models and downstream tasks for source code understanding. In: Proceedings of the 44th International Conference on Software Engineering, 2022. 287–298
7 Liu F, Li G, Fu Z Y, et al. Learning to recommend method names with global context. In: Proceedings of the 44th International Conference on Software Engineering, 2022. 1294–1306
8 Nguyen S, Phan H, Le T, et al. Suggesting natural method names to check name consistencies. In: Proceedings of the ACM/IEEE 42nd international conference on software engineering, 2020. 1372–1384
9 Karampatsis R M, Babii H, Robbes R, et al. Big code!= big vocabulary: Open-vocabulary models for source code. In: Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering (ICSE), 2020. 1073–1085
10 Allamanis M, Peng H, Sutton C. A convolutional attention network for extreme summarization of source code. In: Proceedings of the International Conference on Machine Learning, 2016. 2091–2100
11 Devlin J, Chang M, Lee K, et al. BERT: pre-training of deep bidirectional transformers for language understanding. In: Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, 2019. 4171–4186

12 Raffel C, Shazeer N, Roberts A, et al. Exploring the limits of transfer learning with a unified text-to-text transformer. J Mach Learn Res, 2020, 21: 140
13 Feng Z Y, Guo D Y, Tang D Y, et al. CodeBERT: a pre-trained model for programming and natural languages. In: Proceedings of the Findings of the Association for Computational Linguistics, 2020. 1536–1547
14 Wang Y, Wang W S, Joty S, et al. CodeT5: identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In: Proceedings of the Conference on Empirical Methods in Natural Language Processing, 2021. 8696–8708
15 Ahmad W U, Chakraborty S, Ray B, et al. Unified pre-training for program understanding and generation. In: Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Association for Computational Linguistics, 2021. 2655–2668
16 Wang C Z, Yang Y H, Gao C Y, et al. No more fine-tuning? An experimental evaluation of prompt tuning in code intelligence. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2022. 382–394
17 Hindle A, Barr E T, Su Z D, et al. On the naturalness of software. In: Proceedings of the 34th International Conference on Software Engineering, 2012. 837–847
18 Alon U, Brody S, Levy O, et al. code2seq: generating sequences from structured representations of code. In: Proceedings of the International Conference on Learning Representations (Poster), 2019
19 Zhang J, Wang X, Zhang H Y, et al. A novel neural source code representation based on abstract syntax tree. In: Proceedings of the IEEE/ACM 41st International Conference on Software Engineering, 2019. 783–794
20 Xu X J, Liu C, Feng Q, et al. Neural network-based graph embedding for cross-platform binary code similarity detection. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, 2017. 363–376
21 Allamanis M, Brockschmidt M. Smartpaste: learning to adapt source code. 2017. ArXiv:1705.07867
22 Allamanis M, Brockschmidt M, Khademi M. Learning to represent programs with graphs. In: Proceedings of the International Conference on Learning Representations, 2018
23 Hellendoorn V J, Sutton C, Singh R, et al. Global relational models of source code. In: Proceedings of the International Conference on Learning Representations, 2020
24 Ben-Nun T, Jakobovits A S, Hoefler T. Neural code comprehension: a learnable representation of code semantics. In: Proceedings of the 32nd International Conference on Neural Information Processing Systems, 2018. 3589–3601
25 Sui Y L, Cheng X, Zhang G Q, et al. Flow2Vec: value-flow-based precise code embedding. Proc ACM Program Lang, 2020, 4: 1–27
26 Cummins C, Fisches Z V, Ben-Nun T, et al. ProGraML: a graph-based program representation for data flow analysis and compiler optimizations. In: Proceedings of the 38th International Conference on Machine Learning, 2021. 2244–2253
27 Guo D Y, Ren S, Lu S, et al. GraphCodeBERT: pre-training code representations with data flow. In: Proceedings of the International Conference on Machine Learning, 2021
28 Guo D Y, Lu S, Duan N, et al. UniXcoder: unified cross-modal pre-training for code representation. In: Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics, 2022. 7212–7225
29 Wang X, Wang Y S, Mi F, et al. SynCoBERT: syntax-guided multi-modal contrastive pre-training for code representation. 2021. ArXiv:2108.04556
30 Lewis M, Liu Y, Goyal N, et al. BART: denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In: Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, 2020. 7871–7880
31 VenkataKeerthy S, Aggarwal R, Jain S, et al. IR2VEC: LLVM IR based scalable program embeddings. ACM Trans Archit Code Optim, 2020, 17: 1–27
32 Rozière B, Lachaux M, Chanussot L, et al. Unsupervised translation of programming languages. In: Proceedings of the International Conference on Neural Information Processing Systems, 2020
33 Reimers N, Gurevych I. Sentence-BERT: sentence embeddings using siamese bert-networks. 2019. ArXiv:1908.10084
34 Velickovic P, Cucurull G, Casanova A, et al. Graph attention networks. In: Proceedings of the International Conference on Learning Representations (Poster), 2018
35 Hadsell R, Chopra S, LeCun Y. Dimensionality reduction by learning an invariant mapping. In: Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06), 2006. 1735–1742
36 Oord A v d, Li Y Z, Vinyals O. Representation learning with contrastive predictive coding. 2018. ArXiv:1807.03748
37 Kingma D P, Ba J. Adam: a method for stochastic optimization. 2014. ArXiv:1412.6980
38 Wang W H, Zhang K C, Li G, et al. Learning program representations with a tree-structured transformer. In: Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2023. 248–259
39 Chen M, Tworek J, Jun H, et al. Evaluating large language models trained on code. 2021. ArXiv:2107.03374
40 Nijkamp E, Pang B, Hayashi H, et al. CodeGen: an open large language model for code with multi-turn program synthesis. 2022. ArXiv:2203.13474
41 Cummins C, Petoumenos P, Wang Z, et al. End-to-end deep learning of optimization heuristics. In: Proceedings of the 26th International Conference on Parallel Architectures and Compilation Techniques (PACT), 2017. 219–232
42 Li Y, Tarlow D, Brockschmidt M, et al. Gated graph sequence neural networks. In: Proceedings of the International Conference on Learning Representations (Poster), 2016
43 Kipf T N, Welling M. Semi-supervised classification with graph convolutional networks. 2016. ArXiv:1609.02907
44 He X N, Deng K, Wang X, et al. LightGCN: simplifying and powering graph convolution network for recommendation. In: Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval, 2020. 639–648
45 Wu F, Souza A, Zhang T, et al. Simplifying graph convolutional networks. In: Proceedings of the International Conference on Machine Learning, 2019. 6861–6871
46 Bahri M, Bahl G, Zafeiriou S. Binary graph neural networks. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2021. 9492–9501
47 Wang Y K, Feng B Y, Ding Y F. QGTC: accelerating quantized graph neural networks via GPU tensor core. In: Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2022. 107–119
48 You Y N, Chen T L, Wang Z Y, et al. L2-GCN: layer-wise and learned efficient training of graph convolutional networks. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2020. 2127–2135
49 Chiang W L, Liu X Q, Si S, et al. Cluster-GCN: an efficient algorithm for training deep and large graph convolutional networks. In: Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining,

2019. 257–266
50 Deng X, Zhang Z F. Graph-free knowledge distillation for graph neural networks. 2021. ArXiv:2105.07519
51 Zhang W T, Miao X P, Shao Y X, et al. Reliable data distillation on graph convolutional network. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, 2020. 1399–1414
52 Iyer S, Konstas I, Cheung A, et al. Summarizing source code using a neural attention model. In: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, 2016. 2073–2083
53 Alon U, Zilberstein M, Levy O, et al. code2vec: learning distributed representations of code. Proc ACM Program Lang, 2019, 3: 1–29
54 Peng H, Li G, Wang W H, et al. Integrating tree path in transformer for code representation. In: Proceedings of Advances in Neural Information Processing Systems, 2021. 9343–9354
55 Ma W, Zhao M J, Soremekun E, et al. GraphCode2Vec: generic code embedding via lexical and program dependence analyses. In: Proceedings of the 19th International Conference on Mining Software Repositories, 2022. 524–536
56 Zhang K C, Wang W H, Zhang H Z, et al. Learning to represent programs with heterogeneous graphs. In: Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, 2022. 378–389
57 Cheng X, Nie X, Li N K, et al. How about bug-triggering paths? Understanding and characterizing learning-based vulnerability detectors. IEEE Trans Dependable Secure Comput, 2024, 21: 542–558
58 Liu F, Li G, Zhao Y F, et al. Multi-task learning based pre-trained language model for code completion. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, 2020. 473–485
59 Wan Y, Zhao W, Zhang H Y, et al. What do they capture? A structural analysis of pre-trained language models for source code. In: Proceedings of the 44th International Conference on Software Engineering, 2022. 2377–2388
60 Agarwal P. The cell programming language. Artif Life, 1994, 2: 37–77
61 Cheng X, Zhang G Q, Wang H Y, et al. Path-sensitive code embedding via contrastive learning for software vulnerability detection. In: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, 2022. 519–531
62 Li R, Allal L B, Zi Y T, et al. StarCoder: may the source be with you! 2023. ArXiv:2305.06161