How could Neural Networks understand Programs?

Dinglan Peng [1] **Shuxin Zheng** [2] **Yatao Li** [2] **Guolin Ke** [2] **Di He** [2] **Tie-Yan Liu** [2]

Abstract

Semantic understanding of programs is a fundamental problem for programming language processing (PLP). Recent works that learn representations of code based on pre-training techniques in NLP have pushed the frontiers in this direction. However, the semantics of PL and NL have essential differences. These being ignored, we believe it is difficult to build a model to better understand programs, by either directly applying off-the-shelf NLP pre-training techniques to the source code, or adding features to the model by the heuristic. In fact, the semantics of a program can be rigorously defined by formal semantics in PL theory. For example, the operational semantics, describes the meaning of a valid program as updating the environment (i.e., the memory address-value function) through fundamental operations, such as memory I/O and conditional branching. Inspired by this, we propose a novel program semantics learning paradigm, that the model should learn from information composed of (1) the representations which align well with the fundamental operations in operational semantics, and (2) the information of environment transition, which is indispensable for program understanding. To validate our proposal, we present a hierarchical Transformer-based pre-training model called OSCAR to better facilitate the understanding of programs. OSCAR learns from intermediate representation (IR) and an encoded representation derived from static analysis, which are used for representing the fundamental operations and approximating the environment transitions respectively. OSCAR empirically shows the outstanding capability of program semantics understanding on many practical software engineering tasks. Code and models are released at: https://github.com/pdlan/OSCAR.

[1]University of Science and Technology of China [2]Microsoft Research Asia. Correspondence to: Shuxin Zheng, Yatao Li, Di He <{shuz,yatli,dihe}@microsoft.com>.

1. Introduction

Modern software typically contains tons of code, functions, and modules with overwhelmingly complex structure or organization scheme. It poses great challenges for writing, maintaining, and analyzing such programs. Fortunately, a series of deep learning-based productivity tools were developed to automatically help programmers by analyzing program (Ding et al., 2019; Duan et al., 2020; Yu et al., 2020a), security auditing (Zhou et al., 2019; Buratti et al., 2020), code retrieval (Luan et al., 2019; Ye et al., 2020; Cummins et al., 2020b), and so on.

Inspired by the success of pre-trained representation for semantics understanding of natural language (Devlin et al., 2019; Brown et al., 2020; Xiong et al., 2020), there are many recent attempts to graft the conventional NLP pre-training techniques to source code (Buratti et al., 2020; Feng et al., 2020; Lachaux et al., 2020; Guo et al., 2020; Yu et al., 2020a), in which the code representation is obtained by capturing contextual information from a substantial amount of source code text, and is then used for a variety of downstream software engineering tasks after fine-tuning. For instance, CuBERT (Kanade et al., 2020) leverages the powerful pre-training contextual embedding model BERT (Devlin et al., 2019) to learn informative representations on a Python corpus; CodeBERT (Feng et al., 2020) learns general-purpose representations to bridge natural language (NL) and high-level programming language (PL) by pre-training on NL-PL pairs. Furthermore, features designed by experts (e.g., data flow graph) (Guo et al., 2020) are added to the pre-training model, aiming to provide additional information for program semantics understanding.

However, programming languages have many fundamental differences in essence with natural languages. For example, the same program may exhibit different behaviors against its input and memory state, while there is no such explicit concept in natural language. Therefore, we argue that the current approaches that attempt to capture the semantic proprieties directly from the source code, will limit the semantics understanding of programs, be it applying off-the-shelf NLP pre-training techniques, or adding features to the model by the heuristic.

Indeed, the rigorous mathematical account of the meaning (i.e., the semantics) of programming languages (Gunter,

1992), has been well-studied by formal semantics (Winskel, 1993) in programming language theory. For instance, the operational semantics (van Wijngaarden et al., 2012), which is a widely used branch of formal semantics, captures the meaning of a programming language by defining rules for how its programs execute on an abstract machine. These rules reflect the environment transitions according to the instructions, where the *environment* (Stuart, 2013) is formally defined as a function mapping all memory addresses to their values[1], and one *instruction* conducts a rigorously defined operation, such as reading/writing the memory, basic arithmetic, boolean logic, or conditional branching.

Inspired by the programming language theory, we propose a code representation learning paradigm which could make a model better understand programs. In particular, a code representation should be learned from (1) a translation of the source code text that aligns well with those fundamental operations defined in operational semantics; (2) the information of environment transition, which is obviously indispensable for program understanding.

In order to verify the effectiveness of our proposal, we further present a novel pre-training model called Operational Semantics for Code Abstract Representation (OSCAR) based on a hierarchical Transformer (Vaswani et al., 2017), which is designed to capture the contextual information among long sequences of code. On one hand, to represent the fundamental operations, OSCAR utilizes intermediate representation (IR), which is more applicable for learning code representation rather than high-level programming languages, since the IR is modeled on an abstract machine with a finite instruction set, which can be mapped to the operational semantics almost perfectly. In particular, the IR can be easily acquired by translation from binary or source code of a target program. On the other hand, obtaining concrete and precise information of the environment transition requires plenty of actual executions and calculations, which would be impractical and risky. Therefore, OSCAR alternatively uses abstract information, which can be obtained by abstract interpretation inspired static program analysis without difficulty. Abstract interpretation (Cousot & Cousot, 1977; 1979) describes program semantics by a mathematical characterization of possible behaviors of the program instead of modeling the behaviors after many actual execution trails of the program. In addition, to capture the control structure of a target program or code snippet, we develop a novel Positional Condition Encoding (PCE) to encode the control flow information into the model.

Furthermore, to ensure the desired capacity of pre-trained representation, we design a compact and effective objective function by combining two respective components: a

---

[1]For simplicity, we consider that all the values are stored in memory, e.g., LLVM values.

variant of MLM loss (Devlin et al., 2019) masking entire instructions, and a contrastive loss with different compilation optimization techniques. With instruction tokens as input, the model could capture token-level contextual knowledge by optimizing the variant of MLM loss. Meanwhile, by generating syntactically diverse but functionally equivalent IRs through different compilation optimization techniques, e.g., strength reduction, loop unrolling, and inline expansion, the contrastive loss could provide informative self-supervision, to help the model to efficiently capture the program- or code snippet-level semantic knowledge.

Our contributions are concluded as follows:

- We propose a new learning paradigm that suggests the pre-training model could learn code representation from both the superficial instructions and the underlying environment transitions, which alleviates the aforementioned limitations of semantics understanding of program according to operational semantics.

- We demonstrate our proposal by presenting OSCAR, a hierarchical Transformer which represents the fundamental operations by IR and approximates the environment transitions by an encoded representation derived from static analysis. We also design efficient training objectives for OSCAR to largely facilitate the program semantics understanding.

- OSCAR significantly boosts the performance of semantics understanding of program on a wide range of downstream practical software engineering tasks. Moreover, OSCAR shows remarkable zero-shot ability, i.e., without fine-tuning the parameters, comparing to state-of-the-art pre-training methods.

2. Related Work

Inspired by the great success of deep learning on natural language understanding, there is a growing body of exploratory work on programming language understanding by incorporating code structure into DNN, such as abstract syntax tree (AST) (Alon et al., 2020; Rabinovich et al., 2017; Yin & Neubig, 2017; Wei & Li, 2017; Chen et al., 2018; Alon et al., 2018; Mou et al., 2016; Alon et al., 2019; Zhang et al., 2019; Bui et al., 2020) or graph (Brockschmidt et al., 2018; Wang et al., 2020; Allamanis et al., 2018; Hellendoorn et al., 2019; Duan et al., 2020; Cummins et al., 2020a; Ye et al., 2020; Hellendoorn et al., 2019; David et al., 2020; Wang & Su, 2020).

As the most commonly used architectures in NLP, the Transformer (Vaswani et al., 2017) has also been widely adopted in code understanding tasks. Kim et al. (2020) achieve high accuracy of next token prediction on code by feeding AST to Transformer. Svyatkovskiy et al. (2020) propose to train

a variant of GPT-2 (Radford et al., 2019) from scratch on source code to improve the performance of code completion. Recent works employ pre-training on large-scale code corpus and achieve promising results of code representation. Kanade et al. (2020) pre-train a BERT model on a massive corpus of Python source code and get outstanding performance on five code intelligence tasks. Buratti et al. (2020) present C-BERT, which is a transformer-based model pre-trained on a large collection of corpus written in C. Feng et al. (2020) propose a cross-modal BERT called CodeBERT between source codes and comments, written in programming language and natural language respectively, and gain excellent achievements on NL-PL tasks, such as code search by natural language and code generation from comments. By introducing the data flow to the model, Guo et al. (2020) further improve the performance of CodeBERT. Ahmad et al. (2021) present PLBART, which is also pre-trained on a cross-modal corpus of programming language and natural language via denoising autoencoding. BinaryAI (Yu et al., 2020a) leverage a BERT model pre-trained on binary code to construct a hybrid model by combining with GNN and CNN, and achieves excellent performance on binary code similarity detection. Yu et al. (2020b) further introduce a novel CNN as a feature extractor for source code, and improve the performance of binary-source code matching.

To our best knowledge, the proposed OSCAR is the first attempt for code representation using our PL theory-inspired learning strategy that considers both the superficial programming language and the underlying environment transitions, to improve the performance of program and code understanding.

Intermediate Representation There are prior works (Ben-Nun et al., 2018; VenkataKeerthy et al., 2020; Cummins et al., 2020b) that attempt to understand code on IR language with different motivations. For example, Ben-Nun et al. (2018) argues that training model on a specific source programming language (or machine code for optimization) could not generalize to other languages, and suggests that training on IR language is better since it accepts code in various source languages. Similarly, Cummins et al. (2020b) aims to produce a language-agnostic, compiler-independent representation for the program by leveraging a corpus of LLVM IR covering six source programming languages.

Different from the motivations of previous methods, we suggest that the IR is more applicable for learning code representation rather than high-level PL since the IR is modeled on an abstract machine with a finite instruction set, which can be well mapped to operational semantics.

Contrastive Learning In recent years, contrastive learning (Hadsell et al., 2006; Chen et al., 2020; He et al., 2020) shows promising results on unsupervised visual representation learning. Inspired by this, Jain et al. (2020) present ContraCode, which applies contrastive learning on code representation learning by adopting several source-to-source transformations, such as variable renaming, dead code elimination, and dead code insertion. The functionality of the program would not be changed after the transformations, therefore the underlying representations should be the same.

Different from ContraCode, we generate syntactically diverse but functionally equivalent IRs with different optimization techniques in compilers. Unlike the transformations in ContraCode, different optimizations would lead to huge differences in the syntax and structure of the IR, such as strength reduction, loop unrolling, and inline expansion. This kind of diversity could provide informative self-supervision in helping the model to efficiently capture the program- or code snippet-level semantic knowledge.

3. Method

As mentioned above, operational semantics (van Wijngaarden et al., 2012; Stuart, 2013) captures the meaning of an executable program by the environment transitions according to the instructions on an abstract machine. To be more concrete, we illustrate our motivation by structural operational semantics (Plotkin, 1981; Hennessy, 1990). The meaning of assignment and composition on a simplified abstract machine can be represented respectively as

$$\frac{\langle E, s \rangle \Longrightarrow V}{\langle L := E, s \rangle \to (s \uplus (L \mapsto V))}, \quad \frac{\langle C_1, s \rangle \longrightarrow s'}{\langle C_1; C_2, s \rangle \to \langle C_2, s' \rangle},$$

where E, L, V denote expression, memory location and value respectively, $s \in \mathcal{S}$ denotes environment function mapping all memory locations to values, and C represents code snippet. Therefore, the meaning of assignment can be explained as "the program $L := E$ will update the environment function s with $L = V$ if the expression E in the environment s reduces to V". Similarly, the composition can be explained as "if the code snippet C_1 in environment s finishes in s', then the composed code snippet $C_1; C_2$ in environment s can reduce to execute C_2 in s'".

Obviously, the semantics of a code snippet depends on two parts: the instructions and the information of environment transitions on the abstract machine. Therefore, we propose that a good code representation would be sufficiently learned from these two parts to better understand the semantics of programs. In the following, we will present OSCAR, which is a hierarchical model that learns code representation from these two aspects.
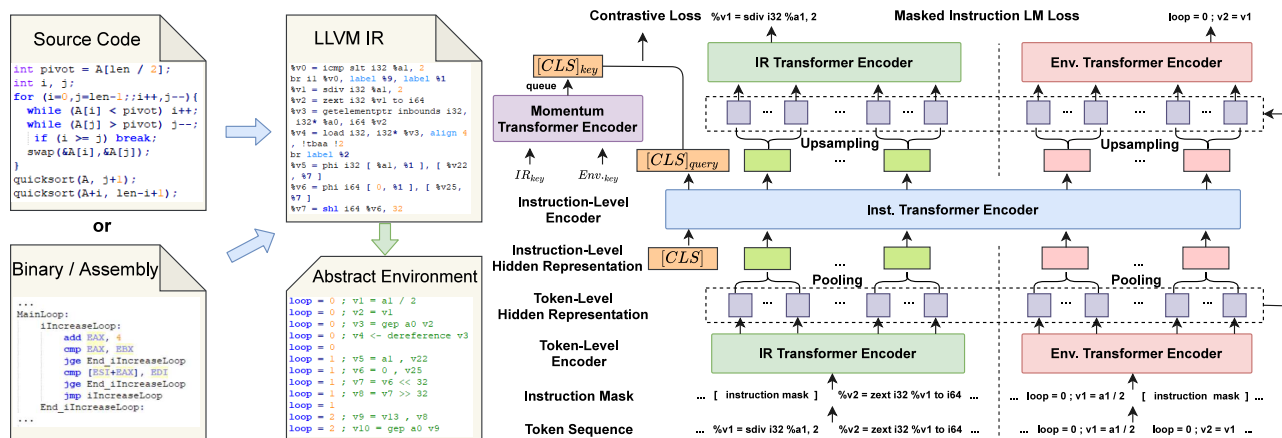
Figure 1. **An illustration of the model architecture of OSCAR.**

3.1. Input Representations

3.1.1. INTERMEDIATE REPRESENTATION (IR)

Learning representation directly from high-level PLs has been widely adopted by existing program understanding methods. However, the gap between the textual representation of source or binary code versus the actual computational meaning becomes larger along with the development of modern programming languages and compilers. This non-negligible gap increases the difficulty of code understanding for existing models.

In general, in order to better analyze and optimize a program, a modern compiler will translate the source code into IR before it generates machine code for a target architecture. IR is modeled after an abstract machine that is typically designed so that each instruction represents exactly one fundamental operation. With this characteristic, IR becomes a more accurate and appropriate representation of the instruction in operational semantics instead of high-level PLs. We collect a large corpus of real-world programs (Details in Appendix F.1) and translate them into LLVM IR as our pre-training data. LLVM IR is one of the most commonly used IR forms, and supports a wide variety of languages.

There is an additional advantage for using IR: if the target code snippet is binary or assembly, the textual information would be easily preserved when translating binary code to IR, unless the binary is generated from strong obfuscation, e.g., executable packing. Meanwhile, translating binary or assembly back to source code (aka. decompilation) would totally change the distribution and legibility of tokens, which would deeply hurt the performance of source code-based methods.

3.1.2. ABSTRACT ENVIRONMENT INFORMATION

We leverage the structural operational semantics to illustrate how we encode the information of environment transitions into the model. The inductive nature of structural operational semantics requires a properly defined initial condition, which is described by the initial environment function. The transitions can then be inferred step-by-step based on the sequencing rules (i.e., composition, transformation, and conditioning, please refer to Plotkin (1981); Hennessy (1990)). To fully capture the concrete and precise information of environment transitions, one has to iterate through many possible combinations of input values and initial conditions, and infer the transitions by actually execute the program with the sequencing rules. This is obviously infeasible since actual executions are quite time-consuming and risky, e.g., analysis for large software projects or malicious software.

Therefore, we alternatively use the abstract environment information obtained from static program analysis, instead of the concrete one. The abstract environment information is inspired by the abstract interpretation (Cousot & Cousot, 1977; 1979), and describes program semantics by a mathematical characterization of possible behaviors of the program instead of modeling the behaviors after many actual execution trails of the program. Applying this idea to structural operational semantics, each expression can reduce to not only a concrete value, but also a relation or a possible range in the value space.

Specifically, we extract three types of relational constraints of the environment from the instructions: those governed by static single assignment (SSA), those by memory reads, and those by memory writes. This information can be easily obtained by LLVM built-in analytic features, e.g., *MemorySSA*. In addition, to better model the range constraints of the environment , we extract auxiliary information from the

control flow graph, i.e., the depth of loop, via LLVM *Loop-Info*. Detailed descriptions about the extraction of abstract environment information can be found in Appendix A.

3.2. Model

3.2.1. ARCHITECTURE

The model architecture of OSCAR is a hierarchical multi-layer Transformer encoder, which is illustrated in Fig.1. In particular, OSCAR consists of two levels of encoders. The lower level is composed of two token-level encoders, which are used to process tokens from IR and abstract environment information, respectively. The upper level is an instruction-level encoder, which aims to extract features further based on the lower-level layer's outputs. The implementation of each level of encoders is identical to BERT (Devlin et al., 2019). We call the two token-level encoders as IR and Env. encoder, and the instruction-level encoder as Inst. encoder.

Typically, the token sequence of a practical program is long. If we simply feed the sequence to a standard Transformer, the time and space cost will be extremely high since the attention module suffers from quadratic computation and memory requirements with respect to the sequence length. Most previous methods truncate the long input sequence (Kanade et al., 2020; Feng et al., 2020) to a short one, such as 512 tokens. But obviously, a 512-long token sequence will lose a significant amount of information in the program or code snippet.

The hierarchical architecture of OSCAR is designed to better solve this problem. We partition the instructions of the input program into groups by every K instructions as one group, and the IR (or abstract environment information) tokens of each group would be fed into parameter-shared IR (or Env.) encoders separately. The output representations coming from one instruction of the token-level encoders, would be averagely pooled, to aggregate the information at the instruction level. Then, those instruction-level hidden representations will be fed to the Inst. encoder for further feature extraction. We set $K = 4$ in our experiments.

Similar to Dai et al. (2020), we up-sample the output sequences of the instruction-level encoder by repeating each hidden vector multiple times so that the length is enlarged to the original token sequence. After up-sampling, consecutive vectors of each instruction would be exactly the same and lost the detailed token-level signals. To involve the uncompressed token-level signal, we adopt a residual connection between uncompressed token-level hidden representations and the up-sampling vectors. After that, another two token-level encoders would try to recover the original token sequences on the positions of the instruction masks.

3.2.2. POSITIONAL CONDITION ENCODING

Since the Transformer is developed to solve the problem of sequence transduction in natural language, it cannot well capture the complicated control structure of programming languages, such as iteration logic and selection logic. However, the control flow information is indispensable for understanding the semantics of a program. To overcome this problem, incorporating the control flow graph (CFG) into Transformer has been widely adopted in prior works (Hellendoorn et al., 2019; David et al., 2020).

In this paper, we design a more simple but effective method called Positional Condition Encoding (PCE), to encode the control flow information into the model through positional encoding. PCE assigns three learnable embedding vectors to the position of each instruction in the target program or code snippet, representing the instruction's current position, and target positions after conditionally jumping with true and false, respectively. Fig.2 shows the illustration of PCE corresponding to the code snippet and the control flow graph, where p_i, p_i^1 and p_i^0 denote the learnable embedding at the current position, true-jumping position, and false-jumping position, of the instruction at position i separately.
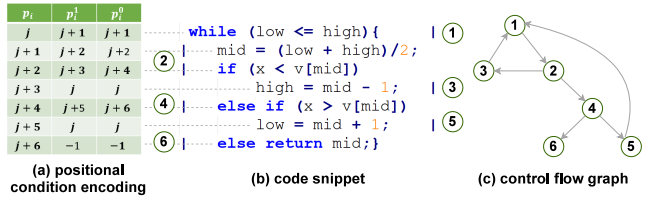

Figure 2. **An illustration of PCE.** PCE could encode the information of control flow graph into the model. Please note that the example code snippet is written in C++ for readability.

To be more concrete, let $h_i \in \mathbb{R}^d$ be the instruction-level hidden representation at position i, and $W^V \in \mathbb{R}^{d \times d}$ denotes learnable projection matrix. The output z_i of the first self-attention module in the Inst. encoder can be written as

$$z_i = \sum_{j=1}^{n} \frac{\exp(\alpha_{ij})}{\sum_{j'=1}^{n} \exp(\alpha_{ij'})} (h_j W^V). \quad (1)$$

Similar to Ke et al. (2020), we propose to model the relationship between positions with different projection matrices. Then, the correlation term α_{ij} in Eq.1 is calculated as

$$\alpha_{ij} = \frac{1}{\sqrt{4d}} (h_i W^Q)(h_j W^K)^T + \frac{1}{\sqrt{4d}} (p_i U^Q)(p_j U^K)^T$$

$$+ \frac{1}{\sqrt{4d}} (p_i^1 U^{1,Q})(p_j^1 U^{1,K})^T + \frac{1}{\sqrt{4d}} (p_i^0 U^{0,Q})(p_j^0 U^{0,K})^T, \quad (2)$$

where $W^Q, W^K \in \mathbb{R}^{d \times d}$ are the projection matrices for the hidden representation h, $U^Q, U^K \in \mathbb{R}^{d \times d}$ are the projection matrices for the current positional embedding p, and $U^{1,Q}, U^{1,K}, U^{0,Q}, U^{0,K} \in \mathbb{R}^{d \times d}$ are for the true-jumping

and false-jumping position embedding p^1 and p^0. The scaling coefficient $\frac{1}{\sqrt{4d}}$ maintains the magnitude of α_{ij}.

From Fig.2 we can see that PCE can incorporate the information about outgoing edges of the nodes in the CFG into the attention module, and the information about incoming edges would also be captured after the calculation of positional correlation in Eq.2. This indicates that OSCAR could capture all information of the CFG with PCE even that the CFG has not been explicitly fed into the model.

3.3. Pre-training Objectives

Masked Instruction LM   Predicting the masked tokens is the most commonly used objective function in previous BERT-based code representation methods (Kanade et al., 2020; Feng et al., 2020; Guo et al., 2020). It's essential to OSCAR that captures the token-level contextual knowledge from optimizing MLM loss during pre-training. However, since both IR and abstract environment information are simultaneously provided to our model, it's trivial to derive particular tokens in the IR through the environment which comes from the same instruction, and vice versa. To prevent such potential information leakage, we propose to mask consecutive tokens of an entire instruction. Specially, we sample randomly 15% instructions from IR and paired environment. We replace the instructions with [MASK] tokens 80% of the time, with random instructions 10% of the time, and leave them unchanged 10% of the time.

Contrastive Learning with Optimization Techniques How to effectively capture the program- or code snippet-level semantic knowledge during pre-training is certainly essential for code representation models. However, it has not been well-studied by prior works.

Actually, modern compilers support versatile compilation options for different demands of optimizations, e.g., minimize execution time, memory footprint, storage size, etc. A single source code snippet could be translated to contrasting IR with different optimization techniques, but the meaning of the code would not be changed. Naturally, the different combinations of multiple optimizations can be used as a method of data augmentation for source code (Details in Appendix E). Motivated by this, we propose to employ an objective on [CLS] token of contrastive learning with a momentum encoder (He et al., 2020) as OSCAR's self-supervised task to better facilitate the semantics understanding from program level, which is illustrated in Fig.1.

4. Experiments

We conduct the pre-training of OSCAR on a large corpus of real-world programs from publicly available open-source GitHub repositories, which covers a broad range of disciplines from operating systems and compilers, to machine learning systems and linear algebra subprograms (Details in Appendix F.1). We evaluate the performance of OSCAR on several semantics understanding tasks for programs in this section. We first perform our model on a practical and important software engineering task, i.e., binary diffing. It is a very fundamental task in reverse engineering and has been widely used to enable different kinds of critical security analysis. After that, we evaluate the performance of OSCAR for high-level PL understanding on the algorithm classification task. Furthermore, as a pre-training method, we investigate the performance of OSCAR in zero-shot learning, where the parameters of OSCAR are fixed. Finally, we analyze the components of our model in the ablation study. Unless otherwise specified, all experiments are conducted on a 12-layer OSCAR model which is composed sequentially of three token-level encoder layers, six instruction-level encoder layers, and three token-level encoder layers. We follow RoBERTa-base (Liu et al., 2019) to set other model configurations (Details in Appendix B), e.g., the dimensionality of hidden representation d is set to 768. The total sequence length of Inst. encoder is set to 512, where the IR and Env. encoders each account for 256 instructions. Detailed descriptions of all downstream datasets and optimization strategies of pre-training and fine-tuning could be found in Appendix G and H respectively.

4.1. Binary Diffing

Binary code differential analysis, a.k.a. binary diffing, is a fundamental analysis capability, which aims to measure the function-level similarity between two given binaries. We evaluate the performance of OSCAR on binary diffing by following the setting and the dataset described in Ding et al. (2019). In addition to Asm2vec (Ding et al., 2019), we further compare OSCAR with two baseline techniques: BinDiff (Dullien & Rolles, 2005), which is the de facto standard binary diffing tool based on graph isomorphism detection; and BinaryAI (Yu et al., 2020a;b), which is a most recently proposed binary code feature extractor based on a hybrid model of neural network with BERT, CNN and GNN, and achieves state-of-the-art performance on code similarity detection.

Following Ding et al. (2019), we evaluate baseline techniques and OSCAR on five commonly used programs using *Recall@1*. All five programs are compiled with GCC 7.5.0 against four different optimization levels. The results are given in Tab.1. As shown, OSCAR consistently outperforms BinDiff, Asm2vec, and BinaryAI across all optimization levels of five programs in terms of recall, by a large margin. For example, in the most difficult matching situation, i.e. diffing between the O0 and O3 optimization levels, OSCAR improves the recall over all baseline techniques on every program.

Table 1. **Binary code similarity detection using the *Recall at position 1 (Recall@1)* metric on popular software across different optimization levels.**

Software	Methods	O0-O1	O0-O3	O1-O3	O2-O3	Avg.
SQLite	BinDiff (Dullien & Rolles, 2005)	0.4360	0.0419	0.1600	0.6455	0.3209
	Asm2vec (Ding et al., 2019)	0.2407	0.2084	0.4270	0.5520	0.2371
	BinaryAI (Yu et al., 2020a;b)	**0.8245**	0.5563	0.6667	0.8107	0.7146
	OSCAR	0.8063	**0.6467**	**0.7148**	**0.8198**	**0.7469**
zlib	BinDiff (Dullien & Rolles, 2005)	0.7143	0.1237	0.1959	0.4271	0.3653
	Asm2vec (Ding et al., 2019)	0.1805	0.2371	0.3814	0.5104	0.3274
	BinaryAI (Yu et al., 2020a;b)	**0.9023**	0.6392	0.7010	0.7708	0.7533
	OSCAR	**0.9023**	**0.7423**	**0.7835**	**0.8229**	**0.8128**
Libcurl	BinDiff (Dullien & Rolles, 2005)	0.5464	0.1893	0.4831	0.8190	0.5095
	Asm2vec (Ding et al., 2019)	0.4911	0.4916	0.6012	0.6426	0.5566
	BinaryAI (Yu et al., 2020a;b)	0.8550	0.7282	0.7991	**0.8620**	0.8111
	OSCAR	**0.8560**	**0.7405**	**0.8190**	0.8512	**0.8167**
BusyBox	BinDiff (Dullien & Rolles, 2005)	0.5364	0.2939	0.6304	**0.9658**	0.6066
	Asm2vec (Ding et al., 2019)	0.3236	0.3767	0.6163	0.6907	0.5018
	BinaryAI (Yu et al., 2020a;b)	0.8541	0.7907	**0.9023**	0.9478	0.8737
	OSCAR	**0.8764**	**0.8183**	0.8883	0.9520	**0.8838**
LibTomCrypt	BinDiff (Dullien & Rolles, 2005)	0.1096	0.0257	0.1768	0.6956	0.2519
	Asm2vec (Ding et al., 2019)	0.4345	0.4319	0.6869	0.7454	0.5747
	BinaryAI (Yu et al., 2020a;b)	0.4906	0.4835	0.6114	0.7491	0.5837
	OSCAR	**0.6483**	**0.5404**	**0.6630**	**0.7583**	**0.6525**

4.2. Algorithm Classification

In this subsection, we study the performance of OSCAR on high-level programming language understanding. We conduct the experiments on POJ-104 dataset (Mou et al., 2016), which contains 104 algorithm problems that were submitted to an online judge system. All samples were written in C/C++ by students. The dataset has around 500 samples per algorithm. The experimental setting we used is exactly same with ProGraML (Cummins et al., 2020a;b), which achieves state-of-the-art classification accuracy on this dataset.

Table 2. **Classification error on POJ-104 test dataset.** The performance of all baseline methods is cited from Cummins et al. (2020b).

Methods	Error(%)
TBCNN (Mou et al., 2016)	6.00
NCC (Ben-Nun et al., 2018)	5.17
XFG (Ben-Nun et al., 2018)	4.56
XFG w/o inst2vec vocab	4.29
ProGraML (Cummins et al., 2020a;b)	3.38
OSCAR	**1.92**

Tab.2 shows the results of classification error. According

to the table, our model achieves significant improvement comparing with all previous methods by a large margin, which indicates that OSCAR could well understand the semantics of source code written in high-level PLs.

4.3. Zero-Shot Learning

In the previous subsection, we show that after fine-tuning the parameters on downstream tasks, OSCAR could outperform prior methods on both binary code or high-level programming language. In this subsection, we further investigate the performance of pre-trained OSCAR in the zero-shot learning setting, i.e., evaluate OSCAR without modifying the parameters. In the comparison, we choose CodeBERT (Feng et al., 2020) as a baseline which shows promising zero-shot ability in the PL-NL probing task. We conduct the empirical study on the code similarity task by leveraging the POJ-104 dataset described above. Following Ye et al. (2020), we label two programs as similar if they are solutions to the same problem, and use mean average precision (MAP) as the evaluation metric. The difference is that we only evaluate our model on the *testing* dataset without using the *training* and *validation* sets.

Since there is no supervision on [CLS] token in CodeBERT during pre-training, it's potentially unfair to only use the representation on [CLS] token in this task. Following Reimers

Table 3. **Mean average precision (MAP) on POJ-104 test dataset.** The performance of all baselines is cited from Ye et al. (2020). The pre-trained model of † is downloaded from the official release of Feng et al. (2020).

	Methods	MAP(%)
Trianing-based	code2vec (Alon et al., 2019)	1.90
	NCC (Ben-Nun et al., 2018)	39.95
	NCC w/o inst2vec	54.19
	Aroma-Dot (Luan et al., 2019)	52.09
	Aroma-Cos	55.12
	MISIM (Ye et al., 2020)	82.45
Pre-training w/o fine-tuning	CodeBERT-[CLS] (Feng et al., 2020)†	10.38
	CodeBERT-avg. of outputs†	9.62
	$OSCAR_{1-6-1}$	45.24
	OSCAR	**49.17**

et al. (2019), we additionally calculate the average of the outputs on all tokens of CodeBERT as the representation in comparison. Furthermore, despite that both CodeBERT and OSCAR have 12 transformer layers, OSCAR has more parameters (163M) than CodeBERT (125M) since there are two simultaneous token-level encoders, i.e., IR encoder and Env. encoder. For a fair comparison, we also report the MAP of a shallow OSCAR with only one token-level encoder layer before and after the six instruction-level encoder layers, which is called $OSCAR_{1-6-1}$ and has only 107M parameters.

As shown in Tab.3, without further modifying the parameters, the pre-trained OSCAR and $OSCAR_{1-6-1}$ both show promising performance on code similarity detection, comparing to other pre-trained models. This indicates that OSCAR has the potential of transferability on downstream tasks without fine-tuning.

Please note, although OSCAR optimizes a similarity loss function (See Sec.3.3) in the pre-training phase, the definitions of two data samples as similar are totally different between pre-training and this task: one labels two IRs as similar if they are generated from the same code snippet with different optimization techniques, and the other labels two programs written by different students as similar if they are solutions to the same OJ problem. Therefore, the objectives of OSCAR pre-training and this task are not the same, and the pre-trained OSCAR model demonstrates the capability of semantics understanding of program in the zero-shot learning setting.

4.4. Ablation Study

In this subsection, we investigate the effects of each component in OSCAR on binary diffing task using BusyBox, and the experimental setting is identical to above. Tab.4 ablates

Table 4. **Ablation study on the components of OSCAR.**

Methods	Avg. Recall@1
OSCAR	**0.8838**
w/o PCE	0.8662
w/o contrastive loss	0.8267
CuBERT (Kanade et al., 2020) w/ IR	0.4650

the effects of the two components of OSCAR: contrastive loss and PCE. As shown in the figure, all components are beneficial, improving the recall on the binary diffing task. Meanwhile, we further train a BERT on IR corpus, which is similar to CuBERT (Kanade et al., 2020) because they share exactly the same architecture, and the only difference is that CuBERT is pre-trained on Python corpus. The result shows that, CuBERT with IR performs not well on the binary diffing task, which reflects the hierarchical architecture of OSCAR is also significantly beneficial.

5. Discussion

In this section, we discuss a few potential drawbacks of our method, which are left for future work.

Real-time Code Analysis Currently, we analyze the target code snippet relying on compiler and static program analysis, which requires that the target code snippet should be compilable. This dependence may limit the applications of OSCAR on real-time code analysis, such as in the modern integrated development environment (IDE). However, there are many alternatives to choose for real-time IR translation and environment information extraction. For example, the interpreter can translate interpreted language (e.g., Python interpreter) into IR in an interactive style; and even for some compiled languages, interactive interpreters are also been developed (e.g., Cling[2] for C++) which support just-in-time (JIT) compilation. With these technologies, there is no need to require the target code snippet to be a compilable program, but only a complete basic block.

Token Semantics Analysis of Source Code When compilers translate source code to IR, partial semantics of tokens is lost since all variables' names would be automatically normalized and replaced by LLVM value identifier. It may lead to a failure of semantics analysis as important information is contained in the variable name. For example, CuBERT (Kanade et al., 2020) claims that it can detect the following code written in Python is buggy:

```
num_batches = batch_size / num_examples
```

where OSCAR may fail in handling this case with high probability. It may be well-solved by keeping the original tokens in IR. We leave it for future work.

---

[2]https://github.com/root-project/cling.

6. Conclusion

In this paper, we propose a novel pre-training model called OSCAR to learn better code representation. Motivated by operational semantics, we suggest that, instead of learning representation directly from high-level programming languages, the intermediate representation is a better abstraction of the semantics of instructions; meanwhile, to well understand the meaning of a program, we propose the abstract environment information should be necessarily considered. Besides, we introduce two additional techniques to make up the OSCAR. First, we incorporate the control flow information into the model through a novel positional encoding called PCE. Second, to provide a code snippet-level self-supervision during pre-training, we introduce contrastive loss by generating syntactically diverse but functionally equivalent IRs with different optimization techniques. OSCAR empirically shows promising results on practical software engineering tasks, including both binary code and high-level programming language understanding, and also demonstrates the transferability on downstream tasks without modifying the parameters of the pre-trained model.

References

Ahmad, W. U., Chakraborty, S., Ray, B., and Chang, K.-W. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333*, 2021.

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

Alon, U., Brody, S., Levy, O., and Yahav, E. code2seq: Generating sequences from structured representations of code. In *International Conference on Learning Representations*, 2018.

Alon, U., Zilberstein, M., Levy, O., and Yahav, E. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

Alon, U., Sadaka, R., Levy, O., and Yahav, E. Structural language models of code. In *International Conference on Machine Learning*, pp. 245–256. PMLR, 2020.

Ben-Nun, T., Jakobovits, A. S., and Hoefler, T. Neural code comprehension: A learnable representation of code semantics. *Advances in Neural Information Processing Systems*, 31:3585–3597, 2018.

Brockschmidt, M., Allamanis, M., Gaunt, A. L., and Polozov, O. Generative code modeling with graphs. *arXiv preprint arXiv:1805.08490*, 2018.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

Bui, N. D., Yu, Y., and Jiang, L. Infercode: Self-supervised learning of code representations by predicting subtrees. *arXiv e-prints*, pp. arXiv–2012, 2020.

Buratti, L., Pujar, S., Bornea, M., McCarley, S., Zheng, Y., Rossiello, G., Morari, A., Laredo, J., Thost, V., Zhuang, Y., et al. Exploring software naturalness throughneural language models. *arXiv preprint arXiv:2006.12641*, 2020.

Chen, T., Kornblith, S., Norouzi, M., and Hinton, G. A simple framework for contrastive learning of visual representations. In *International conference on machine learning*, pp. 1597–1607. PMLR, 2020.

Chen, X., Liu, C., and Song, D. Tree-to-tree neural networks for program translation. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, pp. 2552–2562, 2018.

Cousot, P. and Cousot, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pp. 238–252, 1977.

Cousot, P. and Cousot, R. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pp. 269–282, 1979.

Cummins, C., Fisches, Z. V., Ben-Nun, T., Hoefler, T., and Leather, H. Programl: Graph-based deep learning for program optimization and analysis. *arXiv preprint arXiv:2003.10536*, 2020a.

Cummins, C., Leather, H., Fisches, Z., Ben-Nun, T., Hoefler, T., and O'Boyle, M. Deep data flow analysis. *arXiv preprint arXiv:2012.01470*, 2020b.

Dai, Z., Lai, G., Yang, Y., and Le, Q. V. Funnel-transformer: Filtering out sequential redundancy for efficient language processing. *arXiv preprint arXiv:2006.03236*, 2020.

David, Y., Alon, U., and Yahav, E. Neural reverse engineering of stripped binaries using augmented control flow graphs. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–28, 2020.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT (1)*, 2019.

Ding, S. H., Fung, B. C., and Charland, P. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 472–489. IEEE, 2019.

Duan, Y., Li, X., Wang, J., and Yin, H. Deepbindiff: Learning program-wide code representations for binary diffing. 2020.

Dullien, T. and Rolles, R. Graph-based comparison of executable objects (english version). *Sstic*, 5(1):3, 2005.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

Gunter, C. A. *Semantics of programming languages: structures and techniques*. MIT press, 1992.

Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Yin, J., Jiang, D., et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

Hadsell, R., Chopra, S., and LeCun, Y. Dimensionality reduction by learning an invariant mapping. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, volume 2, pp. 1735–1742. IEEE, 2006.

He, K., Fan, H., Wu, Y., Xie, S., and Girshick, R. Momentum contrast for unsupervised visual representation learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 9729–9738, 2020.

Hellendoorn, V. J., Sutton, C., Singh, R., Maniatis, P., and Bieber, D. Global relational models of source code. In *International conference on learning representations*, 2019.

Hennessy, M. *The semantics of programming languages: an elementary introduction using structural operational semantics*. John Wiley & Sons, 1990.

Jain, P., Jain, A., Zhang, T., Abbeel, P., Gonzalez, J. E., and Stoica, I. Contrastive code representation learning. *arXiv preprint arXiv:2007.04973*, 2020.

Kanade, A., Maniatis, P., Balakrishnan, G., and Shi, K. Learning and evaluating contextual embedding of source code. 2020.

Ke, G., He, D., and Liu, T.-Y. Rethinking the positional encoding in language pre-training. *arXiv preprint arXiv:2006.15595*, 2020.

Kim, S., Zhao, J., Tian, Y., and Chandra, S. Code prediction by feeding trees to transformers. *arXiv preprint arXiv:2003.13848*, 2020.

Lachaux, M.-A., Roziere, B., Chanussot, L., and Lample, G. Unsupervised translation of programming languages. *arXiv preprint arXiv:2006.03511*, 2020.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., and Stoyanov, V. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

Luan, S., Yang, D., Barnaby, C., Sen, K., and Chandra, S. Aroma: Code recommendation via structural code search. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–28, 2019.

Mou, L., Li, G., Zhang, L., Wang, T., and Jin, Z. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016.

Plotkin, G. D. A structural approach to operational semantics. 1981.

Rabinovich, M., Stern, M., and Klein, D. Abstract syntax networks for code generation and semantic parsing. *arXiv preprint arXiv:1704.07535*, 2017.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

Reimers, N., Gurevych, I., Reimers, N., Gurevych, I., Thakur, N., Reimers, N., Daxenberger, J., and Gurevych, I. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2019.

Stuart, T. *Understanding Computation: From Simple Machines to Impossible Programs.* " O'Reilly Media, Inc.", 2013.

Svyatkovskiy, A., Deng, S. K., Fu, S., and Sundaresan, N. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1433–1443, 2020.

van Wijngaarden, A., Mailloux, B. J., Peck, J. E. L., Koster, C. H., Lindsey, C., Sintzoff, M., Meertens, L. G., and Fisker, R. *Revised report on the algorithmic language Algol 68*. Springer Science & Business Media, 2012.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. In *Advances in neural information processing systems*, pp. 5998–6008, 2017.

VenkataKeerthy, S., Aggarwal, R., Jain, S., Desarkar, M. S., Upadrasta, R., and Srikant, Y. Ir2vec: Llvm ir based scalable program embeddings. *ACM Transactions on Architecture and Code Optimization (TACO)*, 17(4):1–27, 2020.

Wang, K. and Su, Z. Blended, precise semantic program embeddings. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 121–134, 2020.

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

Wei, H. and Li, M. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In *IJCAI*, pp. 3034–3040, 2017.

Winskel, G. *The formal semantics of programming languages: an introduction*. MIT press, 1993.

Xiong, R., Yang, Y., He, D., Zheng, K., Zheng, S., Xing, C., Zhang, H., Lan, Y., Wang, L., and Liu, T. On layer normalization in the transformer architecture. In *International Conference on Machine Learning*, pp. 10524–10533. PMLR, 2020.

Ye, F., Zhou, S., Venkat, A., Marucs, R., Tatbul, N., Tithi, J. J., Petersen, P., Mattson, T., Kraska, T., Dubey, P., et al. Misim: An end-to-end neural code similarity system. *arXiv preprint arXiv:2006.05265*, 2020.

Yin, P. and Neubig, G. A syntactic neural model for general-purpose code generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 440–450, 2017.

Yu, Z., Cao, R., Tang, Q., Nie, S., Huang, J., and Wu, S. Order matters: semantic-aware neural networks for binary code similarity detection. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pp. 1145–1152, 2020a.

Yu, Z., Zheng, W., Wang, J., Tang, Q., Nie, S., and Wu, S. Codecmr: Cross-modal retrieval for function-level binary source code matching. *Advances in Neural Information Processing Systems*, 33, 2020b.

Zhang, J., Wang, X., Zhang, H., Sun, H., Wang, K., and Liu, X. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 783–794. IEEE, 2019.

Zhou, Y., Liu, S., Siow, J., Du, X., and Liu, Y. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Advances in neural information processing systems*, 2019.