Article

Not peer-reviewed version

Yul2Vec: Yul Code Embeddings

Krzysztof Fonał *

Posted Date: 3 July 2025

doi: 10.20944/preprints202507.0352.v1

Keywords: Blockchain; Code embeddings; Compilers; Ethereum; Phase-ordering problem; Smart Contracts; Yul

Article

Yul2Vec: Yul Code Embeddings

Krzysztof Fonał

Wrocław University of Science and Technology; krzysztof.fonal@pwr.edu.pl

Abstract

In this paper, I propose Yul2Vec, a novel method for representing Yul programs as distributed embeddings in continuous space. Yul serves as an intermediate language between Solidity code and Ethereum Virtual Machine (EVM) bytecode, designed to enable more efficient optimization of smart contract execution compared to direct Solidity-to-bytecode compilation. The vectorization of a program is achieved by aggregating the embeddings of its constituent code elements from the bottom to the top of the program structure. The representation of the smallest construction units, known as opcodes (operation codes), along with their types and arguments, is generated using knowledge graph relationships to construct a seed vocabulary, which forms the foundation for this approach. This research is important for enabling future enhancements to the Solidity compiler, paving the way for advanced optimizations of Yul and, consequently, EVM code. Optimizing EVM bytecode is essential not only for improving performance but also for minimizing the operational costs of smart contracts - a key concern for decentralized applications. By introducing Yul2Vec, this paper aims to provide a foundation for further research into compiler optimization techniques and cost-efficient smart contract execution on Ethereum. The proposed method is not only fast in learning embeddings but also efficient in calculating the final vector representation of Yul code, making it feasible to integrate this step into the future compilation process of Solidity-based smart contracts.

Keywords: blockchain; code embeddings; compilers; Ethereum; phase-ordering problem; smart contracts; Yul

---

1. Introduction

The Solidity programming language, designed by Gavin Wood under the Ethereum Foundation, was created to facilitate the implementation of smart contracts on the Ethereum Virtual Machine (EVM) using an Object-Oriented paradigm [1]. Initially, Solidity code was compiled directly into EVM bytecode by the Solidity compiler. This bytecode was then deployed onto the blockchain and executed on demand. However, the importance of compiler-level code optimization is a well-established principle in software engineering, and Solidity is no exception to this rule. Direct compilation from Solidity to EVM bytecode posed certain limitations on optimization, prompting the Solidity compiler team to design an intermediate language called Yul. Yul acts as a bridge between Solidity and EVM bytecode, where most of the optimizations are applied [2]. As of today, the Solidity compiler incorporates manually hardcoded sequence of optimization steps.

One of the key challenges associated with compiler optimization is determining the optimal order of optimization steps. This issue, known as phase-ordering problem, is well-known in the domain of compilers and is inherently complex, as the solution space is vast and the problem is classified as NP-complete [3]. Various strategies have been proposed to address this challenge across different programming languages. A common approach involves relying on expert-designed sequences of optimization steps, typically created by the compiler developers or language designers. While this method may not always yield the most efficient solutions, it remains widely used. Notably, the Solidity compiler employs a predefined sequence of optimization steps, as specified by its development team [1].

An early algorithmic approach to address this problem involves the use of genetic algorithms [4–7]. These algorithms evolve random sequences of optimization steps into better-performing chains through iterative refinement. However, genetic algorithms have significant drawbacks. The process of searching for an optimal solution is computationally expensive, and due to the nature of these algorithms, there is no clear separation between the learning phase and the prediction phase. Consequently, this approach often results in a single, static chain of optimization steps. Given the vast diversity of programs, it is virtually impossible to find one universal chain of steps that performs equally well across all use cases. While some programs may benefit from such optimizations, others may experience degraded performance. It is worth noting that in 2020, the Solidity compiler team conducted research to explore the use of genetic algorithms for identifying better optimization sequences [2]. Despite this effort, they ultimately opted to retain a manually curated optimization sequence.

The introduction of neural networks marked a significant milestone in the quest to find solutions for compiler optimization with distinct learning and prediction phases. In [8], the authors proposed NEAT, an approach that combines genetic algorithms and neural networks to predict dynamic, program-specific optimization steps. Unlike static optimization sequences, NEAT enables the generation of tailored steps for individual programs and even specific functions. This method was implemented and tested on a Java compiler, yielding promising results. With the advent of greater computational power and advancements in deep neural network techniques, researchers began exploring reinforcement learning (RL) as a natural and elegant solution to the problem of determining optimization steps.

Reinforcement learning has since become a focal point of research in this domain. For example, in [9,10], RL-based approaches were proposed to address the phase-ordering problem in LLVM. These studies compared RL-based solutions to the standard LLVM compiler optimization level (-O3), demonstrating the potential for RL to outperform conventional methods. The author of these works suggest that future advancements in this field may involve hybrid solutions combining reinforcement learning and transformer models, which are seen as a promising direction for solving optimization challenges more effectively.

However, applying neural networks to this problem introduces a significant challenge: program code must be represented in the form of constant-sized vectors. This requires the development of methods to vectorize program code or its components (e.g., functions). While several studies have proposed solutions for vectorizing LLVM's Intermediate Representation (IR), such as [11,12], no equivalent solution currently exists for Yul, the intermediate language used in the Solidity compiler. Addressing this gap is the primary focus of this paper.

The lack of a publicly available dataset of Yul code compounds the difficulty of this task. While datasets for Solidity code are available, the optimization improvements discussed earlier require embeddings specifically derived from Yul code. To address this, the author first created a comprehensive dataset of approximately 350,000 Yul scripts [13]. Drawing inspiration from [12], the author generated triplets for knowledge graph embedding, which served as embedding seeds for Yul elements. Finally, a solution was developed to calculate vector representations for Yul-based programs. This method computes the final vector representation of a Yul program by summing the vectors of all components within the program's Abstract Syntax Tree (AST).

One of the key challenges the author encountered was validating the results, which arises for two main reasons. First, as previously noted, to the best of the author's knowledge, there are no prior attempts to vectorize Yul code. This lack of prior work means there are no existing results or benchmarks for direct comparison. Second, evaluating the quality of the obtained vector representa-

[1] Default optimization steps settled by Solidity team can be found at https://github.com/ethereum/solidity/blob/develop/libsolidity/interface/OptimiserSettings.h#L42

[2] Report from this research can be found at https://github.com/ethereum/solidity/issues/7806

tions is inherently complex. How can one definitively judge whether the generated vectors accurately represent the structure and relationships of Yul programs?

Manually assessing the similarity between thousands of Yul codes to validate the vector distribution is not feasible. Human judgment alone cannot reliably determine how "close" certain programs are to one another in terms of their structural or functional characteristics. To address this, the author made a best-effort attempt by categorizing contracts into meaningful groups and analyzing whether the resulting vector distributions aligned with these categories. While not definitive, this approach provides some evidence of the method's effectiveness. Additionally, the distribution of embedding seeds was analyzed, and their alignment with expectations further supports the plausibility of the generated vectors.

However, the most rigorous way to validate the proposed vectorization method would be to integrate it into a practical application - such as combining it with a neural network-based phase-ordering solution like reinforcement learning. The success of such an approach would provide a more concrete and objective measure of the vectorization method's effectiveness. This step, however, is beyond the scope of the current work and is left as a topic for future research.

The structure of this paper is as follows. Section 2 reviews related work, focusing on prior attempts to address similar problems for different programming languages and compilers. Additionally, it examines foundational research that underpins the proposed solution, including studies on graph embeddings. Section 3 provides a deeper theoretical background to knowledge graph embeddings, and also it introduce to the grammar and structure of the Yul intermediate language, and other relevant concepts. Section 4 outlines the proposed solution for vectorizing Yul code. It describes the methodology, implementation details, and techniques used to generate vector representations for Yul-based programs. The results achieved through the implementation of this approach are presented and analyzed in Section 5, highlighting its effectiveness and key observations. Finally, Section 6 concludes the paper by summarizing the contributions and discussing potential applications and future directions for research in this area.

2. Related Works

The representation of source code for machine learning applications has evolved significantly in recent years. Traditional compiler-based methods - such as representing programs as lexical tokens, Program Dependence Graphs (PDGs), or Abstract Syntax Trees (ASTs) [14–17] - are well suited for parsing, compilation, and static analysis. However, these symbolic representations are generally incompatible with modern AI techniques, which rely on dense, continuous vector inputs to train deep neural networks. To address this limitation, researchers have adopted code embeddings inspired by word embedding techniques in natural language processing, such as Word2Vec by Mikolov et al. [18] and GloVe by Pennington et al. [19]. Code embeddings aim to encode both the semantic and syntactic properties of programs into high-dimensional vectors, facilitating similarity comparisons and supporting downstream tasks such as code classification, summarization, and repair.

Early efforts to vectorize source code leveraged structured representations such as ASTs. For example, code2vec by Alon et al. [11] introduced a neural architecture that learns embeddings for Java code by aggregating paths in the AST, capturing both syntactic and semantic relationships to support tasks like method name prediction and code summarization. This approach was later extended by code2seq [20], which generates vector sequences to improve interpretability. Although AST-based methods excel at encoding hierarchical code features, they often struggle to scale effectively for large codebases and to capture complex dependencies. Kanade et al. developed CuBERT [21], a BERT-based model pre-trained on Python source code that leverages contextual embeddings to capture language-specific semantics. Similarly, Mou et al. [22] proposed a tree-based convolutional neural network (CNN) to classify C++ programs by embedding ASTs, while Gupta et al. [23] introduced a token-based sequence-to-sequence model to diagnose and repair common errors in student-written C programs.

These approaches highlight the importance of domain-specific adaptations but are typically limited in their generalizability across different programming languages.

Another notable study proposes a generic triplet-based embedding framework [12]. Unlike the previously mentioned approaches, which are language-specific, this method is designed to be language-agnostic. In this framework, code elements are decomposed into structural "triplets" (entity-relation-entity) and subsequently encoded using a Knowledge Graph Embedding method [24,25]. However, while the embedding technique itself is language-independent, generating the triplets requires language-specific preprocessing. The authors demonstrated this approach using LLVM Intermediate Representation (IR), a stack-based low-level language (similar to assembly) that simplifies triplet extraction due to its linear structure. Their method also includes a process for generating a complete vector representation of a program from the IR embeddings.

Building on these insights and the aforementioned triplet-based framework, this study proposes a novel method to embed Yul, an intermediate language used in Ethereum smart contracts. Unlike lower-level languages, Yul is a higher-level language that incorporates constructs such as if-statements, switch statements, continue statements, functions, and other control structures, and its structure can be represented as an AST. Thus, the tree structure must be analyzed to generate the required triplets for obtaining embeddings of Yul operations. In a subsequent phase of AST analysis, these embeddings are aggregated into a holistic vector representation of the entire program. This complete method is referred to as yul2vec.

3. Background

3.1. Ethereum and Solidity

The emergence of the Ethereum blockchain in 2014 marked a significant advancement in the field of cryptocurrencies, representing the most substantial innovation since the deployment of Bitcoin.

Bitcoin's core concept revolves around a decentralized, distributed ledger secured through cryptographic techniques, which enables the tracking of transactions between parties without reliance on a central authority. Ethereum extended this fundamental idea by introducing the capability to not only track asset flows but also maintain the state of applications within its ledger. This enhancement enables applications to utilize a globally distributed storage system where data is immutable and cannot be altered or corrupted by third parties.

This breakthrough paved the way for a new and expansive market for so-called *smart contracts*—decentralized, distributed applications that leverage Ethereum's global storage. To facilitate the development of smart contracts on the Ethereum Virtual Machine (EVM), the Ethereum Foundation introduced Solidity, a high-level programming language designed specifically for this purpose [26]. Solidity shares some structural similarities with JavaScript, making it accessible to developers familiar with web-based scripting languages.

Contracts written in Solidity are compiled into EVM bytecode, which is then deployed onto the Ethereum blockchain. The cost of deploying a smart contract depends on the amount of bytecode stored in the blockchain, as measured by its byte size. Additionally, executing any smart contract function that modifies the blockchain state incurs a extitgas fee.

3.2. Yul Intermediate Layer

As with many programming languages, Solidity employs various optimizations to enhance the efficiency of compiled code. However, unlike conventional compilers that primarily optimize for specific CPU architectures or execution speed, Solidity optimizations serve a dual purpose. The first objective is to minimize the size of the deployed bytecode, particularly the *initialization code*, which is functionally similar to constructors in object-oriented programming languages. The second objective is to optimize gas consumption, thereby reducing the cost of smart contract execution.

Directly optimizing Solidity code into EVM bytecode proved to be a challenging and inefficient task. To address this issue and enhance the effectiveness of bytecode optimizations, the Solidity team introduced an intermediate representation language known as Yul in 2018. Over several years of development, Yul matured and was officially recognized as a standard, production-ready, intermadiate, Solidity compilation step by 2022.

Yul operates at the level of EVM opcodes while also maintaining features reminiscent of high-level programming languages. It includes control structures such as loops, switch-case statements, and functions, distinguishing it from low-level assembly languages or intermediate representations like LLVM IR. Despite its expressive syntax, Yul maintains a relatively simple grammar. Listing 1 provides a syntax specification, which serves as the foundation for extracting code features and vectorizing Yul code for further analysis. A formal grammar specification for Yul can be found in the official documentation [27]. A distinctive characteristic of Yul is its strict adherence to a single data type—the 256-bit integer—aligning with the native word size of the EVM. This constraint simplifies the language while ensuring compatibility with the underlying EVM infrastructure.

It is also important to elaborate on the structure of Yul programs, which actually consist of two distinct components. When Solidity code is compiled, it produces a Yul representation in the form of an Abstract Syntax Tree (AST). This AST contains two subcomponents: the initialization code and the runtime code. The first component—the initialization code—is executed at the time the smart contract is deployed. Its purpose is to set up the necessary storage variables and initialize them with default or specified values. This initialization logic can be non-trivial and is somewhat analogous to the constructor of an object in object-oriented programming languages. The second component—the runtime code—represents the core logic of the smart contract. This code is stored on the blockchain and is what actually handles interactions after deployment. Figure 1 illustrates an overview of the Yul architecture.

Listing 1. Yul specification schema.

```
Block = '{' Statement* '}'
Statement =
    Block |
    FunctionDefinition |
    VariableDeclaration |
    Assignment |
    If |
    Expression |
    Switch |
    ForLoop |
    BreakContinue |
    Leave
FunctionDefinition =
    'function' Identifier '(' TypedIdentifierList? ')'
    ( '->' TypedIdentifierList )? Block
VariableDeclaration =
    'let' TypedIdentifierList ( ':=' Expression )?
Assignment =
    IdentifierList ':=' Expression
Expression =
    FunctionCall | Identifier | Literal
If =
    'if' Expression Block
Switch =
    'switch' Expression ( Case+ Default? | Default )
Case =
    'case' Literal Block
Default =
    'default' Block
```

```
ForLoop =
    'for' Block Expression Block Block
BreakContinue =
    'break' | 'continue'
Leave = 'leave'
FunctionCall =
    Identifier '(' ( Expression ( ',' Expression )* )? ')'
Identifier = [a–zA–Z_$] [a–zA–Z_$0–9.]*
IdentifierList = Identifier ( ',' Identifier)*
TypeName = Identifier
TypedIdentifierList = Identifier ( ':' TypeName )? ( ',' Identifier ( ':' TypeName )? )*
Literal =
    (NumberLiteral | StringLiteral | TrueLiteral | FalseLiteral) ( ':' TypeName )?
NumberLiteral = HexNumber | DecimalNumber
StringLiteral = '"' ([^"\r\n\\] | '\\' .)* '"'
TrueLiteral = 'true'
FalseLiteral = 'false'
HexNumber = '0x' [0–9a–fA–F]+
DecimalNumber = [0–9]+
```

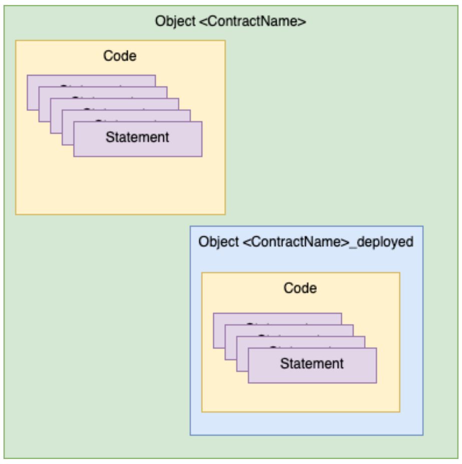


Figure 1. Yul high-level overview. The main object in Yul's Abstract Syntax Tree (AST) consists of a code block, which includes the contract's initial code and a nested object containing the code that runs once the contract is deployed. The code block contains a list of statements, where each statement can represent various constructs.

3.3. Importance of Yul Optimization

Compiler optimizations are important for enhancing the efficiency and performance of applications, particularly because automated techniques applied at different levels (closer to binary code, at the intermediate language level, etc.) can often outperform manual optimizations. In the case of Yul, the primary objective of optimization is to reduce the cost of Ethereum transactions.

Transactions on the Ethereum blockchain can take various forms. Similar to Bitcoin, transactions may involve direct cryptocurrency transfers. However, Ethereum also supports transactions that invoke smart contract functions, modifying their internal state. The sender bears the cost of these transactions in the form of gas fees, making cost-effectiveness a critical factor in the adoption and usability of decentralized applications.

In contrast to Bitcoin's non-Turing-complete scripting language—where execution costs are directly proportional to script size - Ethereum's Turing-complete smart contracts introduce additional complexity. Due to the presence of loops and dynamic computations, execution costs cannot be determined in advance. To address this, Ethereum employs a *gas* model, in which each EVM opcode has a predefined gas cost. The total execution cost of a smart contract function is calculated as the sum of the gas costs of the executed opcodes.

Given the direct correlation between bytecode efficiency and transaction costs, optimizing Yul code has a significant impact on reducing gas fees. Efficient bytecode minimizes computational overhead, making decentralized applications more affordable and accessible to users. The bytecode size is also important, as it directly affects the initial deployment cost of the contract (i.e., the cost of storing the script on the blockchain). However, this deployment cost is a one-time expense, making it less critical compared to the recurring costs associated with contract execution. Table 1 provides an overview of various EVM opcodes and their corresponding gas costs, highlighting the importance of compiler-driven optimizations in reducing transaction expenses.

Table 1. Some of EVM Opcodes.

Opcode Number	Opcode Name	Minimum Gas	Description
0x00	STOP	0	Halts execution
0x01	ADD	3	Addition operation
0x02	MUL	5	Multiplication operation
0x03	SUB	3	Subtraction operation
0x04	DIV	5	Integer division operation
0x06	MOD	5	Modulo operation
0x0a	EXP	10	Exponential operation
0x10	LT	3	Less-than comparison
0x11	GT	3	Greater-than comparison
0x12	SLT	3	Signed less-than comparison
0x13	SGT	3	Signed greater-than comparison
0x14	EQ	3	Equality comparison
0x15	ISZERO	3	Simple not operator
0x16	AND	3	Bitwise AND operation
0x17	OR	3	Bitwise OR operation
0x18	XOR	3	Bitwise XOR operation
0x19	NOT	3	Bitwise NOT operation
0x1a	BYTE	3	Retrieve single byte from word
0x30	ADDRESS	2	Get address of currently executing account
0x31	BALANCE	100	Get balance of the given account
0x32	ORIGIN	2	Get execution origination address
0x33	CALLER	2	Get caller address
0x36	CALLDATASIZE	2	Get size of input data
0x37	CALLDATACOPY	3	Copy input data to memory
0x3a	GASPRICE	2	Get price of gas in current environment
0x3b	EXTCODESIZE	100	Get size of an account's code
0x3c	EXTCODECOPY	100	Copy an account's code to memory
0x3f	EXTCODEHNSH	100	Get hash of an account's code
0x41	COINBASE	2	Get the block's beneficiary address
0x42	TIMESTAMP	2	Get the block's timestamp
0x43	NUMBER	2	Get the block's number
0x45	GASLIMIT	2	Get the block's gas limit
0x50	POP	2	Remove item from stack
0x51	MLOAD	3	Load word from memory
0x52	MSTORE	3	Save word to memory
0x53	MSTORE8	3	Save byte to memory
0x54	SLOAD	100	Load word from storage
0x55	SSTORE	100	Save word to storage
0x56	JUMP	8	Alter the program counter
0x57	JUMPI	10	Conditionally alter the program counter

3.4. Representation Learning

As previously explained, the objective of this paper is to obtain vector representations of Yul programs to serve as suitable inputs for various machine learning tasks. To develop a method for vectorizing entire programs, it is necessary to establish a structural representation of the atomic components of the Yul language, particularly individual instructions.

To achieve meaningful representations of individual instructions, machine learning techniques are employed. Given the absence of labeled data for supervised learning - since generating such data is the very task at hand - unsupervised learning emerges as the natural choice. An unsupervised method must autonomously learn to represent individual instructions by capturing their relationships, similarities, and distinctive features. One promising approach for this is the use of context-window-based embeddings, as seen in models like Word2Vec. These embeddings capture semantic similarities by analyzing the local context of words or tokens within a fixed window. While effective in identifying local syntactic patterns, such embeddings may fall short in capturing the broader structural relationships inherent in programming languages.

A more suitable alternative is Knowledge Graph Embeddings (KGEs), which model relationships between entities and their interactions. In this approach, knowledge is represented as a collection of triplets in the form $< h, r, t >$, where h (head) and t (tail) are entities, and r represents the relation between them. The goal is to learn vector representations of these components, where relationships are modeled as *translations* in a high-dimensional space. Figure 2 illustrates how such translations can be represented in two-dimensional space. This translational property forms the foundation of many KGE methods, which operate under the assumption that, given any two elements of a triplet, it should be possible to infer the third. Various methods accomplish this in different ways.

In this paper, I employ the TransE method [28], a well-known KGE approach. TransE learns representations by modeling relationships in the form $h + r \approx t$ for a given triplet $< h, r, t >$. It achieves this by being trained on a series of valid triplets while minimizing a *margin-based ranking* loss $\mathcal{L}$, which ensures that distance of the vector representation of a valid triplet $< h, r, t >$ between vector of the invalid triplet $< h', r, t' >$ are at least separated by a margin m. The loss function is given by:

$$\mathcal{L} = \sum_{<h,r,t>} \sum_{<h',r,t'>} [m + distance(h + r, t) - distance(h' + r, t')]_+$$

, where $[]_+$ denotes the hinge loss.

The TransE method is computationally efficient, requiring relatively few parameters, and has been shown to be both effective and scalable [29].

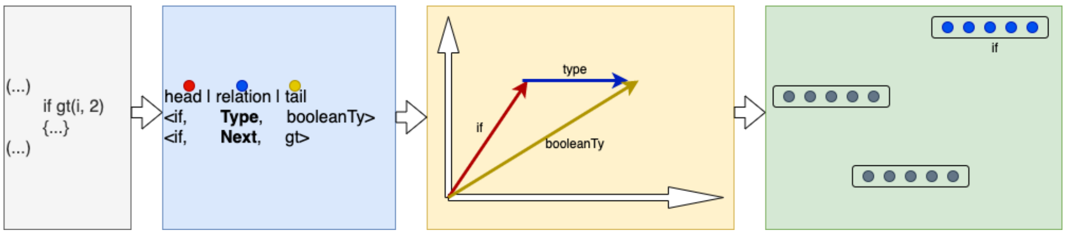


Figure 2. This figure shows example of processing representation learning. The leftmost image shows a sample line of Yul code being processed. The instruction - *if* in this case - is analysed in order to create a triplets from it. In this case *if* has *Type* relation with booleanTy (i.e. *if* is type of boolean) and *Next* relation with *gt* function (i.e. *gt* is next instruction right after *if*). There is also third type of relation Arg1..ArgN for arguments of given instruction (if exists). KGE method then attempts to determine vector representations, ensuring that when the head entity is summed with the relation, it lands close to the tail entity. After the learning process, all extracted atomic elements of Yul are assigned vector representations, forming a structured vocabulary.

4. Yul Code Vectorization

In this chapter, I present the proposed method for vectorizing Yul code. First, I discuss the process of learning embeddings for atomic entities, followed by embeddings for entire Yul contracts, which are obtained through an analysis of the abstract syntax tree (AST) of the Yul code.

I believe this work will initiate research on applying AI to Yul code and the Solidity compiler.

4.1. Vectorizing Atomic Yul Entities

The first step involves learning embeddings for the atomic components of Yul code - namely, the entities used in representation learning, as described in Section 3.4. As previously mentioned, this work is based on the framework presented in [12], where the authors extracted entities and learned embeddings using the Knowledge Graph Embedding (KGE) method for the LLVM-IR language. In this study, the initial task was to define entities specific to the Yul language.

According to the framework, three categories of entities are required: operations (opcodes), types of operations, and argument variations. As mentioned in Section 3.2, Yul includes built-in opcodes that correspond to EVM opcodes, which naturally fall into the first category. Additionally, developers can define custom functions. In this work, all such function calls are represented by a single custom opcode named *functioncall*. Furthermore, unlike simple stack-based languages, Yul provides a basic high-level structure with elements commonly found in Java, JavaScript, C++, and other high-level languages, such as *for* loops and *switch* statements. These keyword-based statements are also considered separate opcodes.

The second category, types of operations, presents a challenge because Yul is based on a single data type, a 256-bit value. To introduce differentiation among opcode types and improve vector distribution, an artificial classification was proposed. For example, boolean instructions that return only a 0 or 1 (as a 256-bit value) are categorized as the *boolean* type. Similarly, operations that handle address-like data (such as blockchain addresses) are classified under the *address* type.

The third category pertains to variations in opcode arguments. An argument may be a literal, a variable, or function call. Table 2 lists the extracted entities used for embedding learning (EVM opcodes are omitted, as they were discussed in previous sections and tables). The relationships between entities include *Type* (type of entity), *Next* (next instruction), and *Arg0...ArgN* (*n*-th argument). Figure 3 illustrates example triplets extracted from several lines of code.

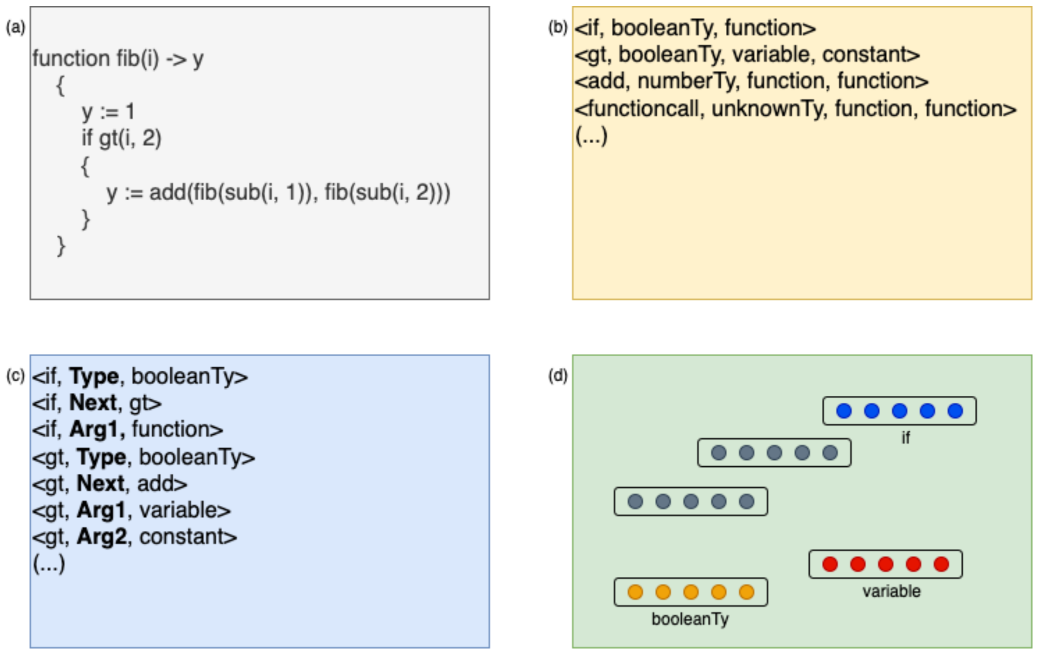
Table 2. Entities in the Embedding Space.

Opcode Entities	Type Entities	Argument Entities
break	addressTy	function
forloop	booleanTy	variable
functioncall	numberTy	constant
if	voidTy	
leave	unknownTy	
return		
switch		
...		

With a defined schema for entities, the next step is to process Yul code to collect entities for the KGE learning process. As previously mentioned, Yul is not a stack-based language, meaning entity extraction is not as straightforward as iterating through the code line by line. Instead, we must obtain the Abstract Syntax Tree (AST) representation of the program. The Solidity compiler (*solc*) provides an option to generate an AST in JSON format from Yul code. Using this representation, I apply graph processing with a depth-first search (DFS) approach, adapting the traversal method to the specific node types defined by Yul's syntax.

For example, a *for-loop* in Yul consists of multiple components: a pre-condition (where the loop variable may be defined), a condition (e.g., `x < 10`), a post-condition (where the loop variable can be modified), and the loop body (which contains the instructions executed in each iteration). Extracting triplets from a *for-loop* requires processing each of these components separately. The loop body itself may contain a sequence of statements, each of which might introduce additional structures requiring deeper traversal.

Listing 2 presents a pseudocode implementation for extracting triplets from Yul code.



(a)

```
function fib(i) -> y
  {
      y := 1
      if gt(i, 2)
      {
          y := add(fib(sub(i, 1)), fib(sub(i, 2)))
      }
  }
```

(b)

```
<if, booleanTy, function>
<gt, booleanTy, variable, constant>
<add, numberTy, function, function>
<functioncall, unknownTy, function, function>
(...)
```

(c)

```
<if, Type, booleanTy>
<if, Next, gt>
<if, Arg1, function>
<gt, Type, booleanTy>
<gt, Next, add>
<gt, Arg1, variable>
<gt, Arg2, constant>
(...)
```

Figure 3. This figure illustrates the process of triplet preparation for subsequent KGE processing. In (a), a sample of Yul code is shown. This code is represented in AST form and traversed to extract tuples, as depicted in (b). Each tuple consists of an instruction, its type, and a list of argument types. The next step is to transform this list of tuples into triplets (c), as described in Figure 2. Ultimately, this process generates the fundamental vocabulary (d) required for vectorizing entire Yul contracts.

4.2. Vectorizing Entire YUL Contracts

Although Yul is a high-level language constructed and represented in an Abstract Syntax Tree (AST) form, at the lowest level of the representation, the fundamental unit that undergoes vectorization is a single expression. According to the Yul specification, such an expression can be categorized as a function call, an identifier (i.e., a variable name), or a literal (e.g., a constant value such as *10*).

The vectorization process relies on previously categorized embeddings, which include EVM opcodes, custom opcodes, return types, and argument types. The formal definition for vectorizing a single expression is given as follows:

$$\mathbf{V}(E^{(n)}) = O^{(n)} \cdot w_o + T^{(n)} \cdot w_t + \left(\sum A_{(i)}^{(n)}\right) \cdot w_a, \tag{1}$$

where $\mathbf{V}$ denotes the vectorization function, and $E^{(n)}$ represents an expression indexed by n. The terms $O^{(n)}$ and $T^{(n)}$ correspond to the opcode and return type vectors of the given expression, which are obtained from previously trained embeddings. The term $A_{(i)}^{(n)}$ represents the argument vectors, which are computed in a context-aware manner and will be described in detail in the following section. The individual components are weighted by the factors w_o, w_t, and w_a, respectively.

Notably, some opcodes or custom functions do not have associated arguments. In such cases, the last term in Equation (1) becomes a zero vector. For example, Yul keywords that are mapped directly to opcodes, such as *break* and *leave*, are treated as simple opcodes with an associated return type of *voidTy*:

$$\mathbf{V}(E_{\text{break}}) = O^{[\text{break}]} \cdot w_o + T^{[\text{voidTy}]} \cdot w_t,$$

$$\mathbf{V}(E_{\text{leave}}) = O^{[\text{leave}]} \cdot w_o + T^{[\text{voidTy}]} \cdot w_t$$

On the other hand, conditional expressions within control flow statements, such as *if* and *for-loop*, are treated as arguments. Specifically, the condition expression in an *if* statement is vectorized as follows:

$$\mathbf{V}(E_{\text{if}}) = O^{[\text{if}]} \cdot w_o + T^{[\text{booleanTy}]} \cdot w_t + \mathbf{V}(E_{\text{if-condition}}) \cdot w_a.$$

This approach ensures that expressions within control flow statements maintain structural consistency during the vectorization process. In particular, the weighting factors (w_o, w_t, and w_a) play a crucial role in capturing the relative importance of opcodes, return types, and arguments, respectively. The subsequent section provides a deeper discussion on the context-aware computation of argument vectors, which further refines the expressiveness of the vectorized representations.

Listing 2. Triplets extraction from Yul code.

```
function extract_arguments(ast):
    arguments = []
    for arg in ast:
        if arg["nodeType"] eq "YulFunctionCall":
            arguments += "function"
        else if arg["nodeType"] eq "YulIdentifier":
            arguments += "variable"
        else if arg["nodeType"] eq "YulLiteral":
            arguments += "constant"
        else:
            arguments += "unknown"
    return arguments

function extract_function_call(ast):
    opcode_name = ast["functionName"]["name"]
    opcode_type = EVM_OPCODES[ast["functionName"]["name"]]
    if opcode_type is null:
        opcode_type = "unknownTy"
        opcode_name = "functioncall"

    append_triplet(opcode_name, opcode_type, extract_arguments(ast['arguments']))

function extract_triplets(ast):
    nodeType = ast["nodeType"]

    match nodeType:
        case "YulObject":
            extract_triplets(ast["code"])
        case "YulCode":
            extract_triplets(ast["block"])
        case "YulBlock":
            for statement in ast["statements"]:
                extract_triplets(statement)
        case "YulFunctionCall":
            extract_function_call(ast)
        case "YulExpressionStatement":
            extract_triplets(ast["expression"])
        case "YulVariableDeclaration" | "YulAssignment":
            extract_triplets(ast["value"])
        case "YulFunctionDefinition":
            extract_triplets(ast["body"])
        case "YulIf":
            append_triplet("if", "booleanTy", extract_arguments([ast['condition']]))
            extract_triplets(ast["body"])
        case "YulForLoop":
            append_triplet("forloop", "voidTy")
            extract_triplets(ast["pre"])
            extract_triplets(ast["condition"])
            extract_triplets(ast["post"])
            extract_triplets(ast["body"])
        case "YulSwitch":
            append_triplet("switch", "voidTy", extract_arguments([ast['expression']]))
            for switch_case in ast["cases"]:
                extract_triplets(switch_case["body"])
        case "YulBreak":
            append_triplet("break", "voidTy")
        case "YulContinue":
            append_triplet("continue", "voidTy")
        case "YulLeave":
            append_triplet("leave", "voidTy")
        case "YulIdentifier" | "YulLiteral":
            # do nothing

for yul_code in yul_dataset:
    ast = parse_ast(yul_code)
    extract_triplets(ast)
```

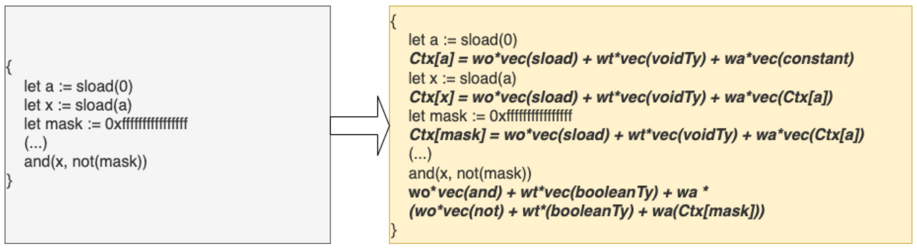
4.3. Context Awareness

The notion of *context awareness* in the vectorization of Yul code can be understood through two key approaches.

The first approach involves handling non-EVM built-in functions. Whenever a custom function (i.e., a function that is not part of the learned embeddings) is invoked, its opcode vector is not directly assigned. Instead, the vector representation of the function is first computed based on its definition. This ensures that custom functions are evaluated dynamically at runtime rather than relying on a generic *functioncall* vector. As a result, vectorized expressions are more precise and are tailored to the specific execution context of the program.

The second crucial aspect of context awareness is **variable tracking**. A variable definition (i.e., a declaration without an immediate assignment) is represented as a zero vector since it does not yet hold any meaningful computational value. However, once an assignment occurs, the computed vector of the assigned expression is stored. Whenever the variable is subsequently used as an argument in another expression, its stored vector value is substituted accordingly. This mechanism ensures that variables retain contextual information across expressions, improving the fidelity of the vectorized representation.

An illustration of this variable tracking mechanism is presented in Figure 4, which demonstrates how assigned values propagate through vectorized expressions while preserving contextual dependencies.



```
{
   let a := sload(0)
   let x := sload(a)
   let mask := 0xffffffffffffffff
   (...)
   and(x, not(mask))
}
```

```
{
   let a := sload(0)
   Ctx[a] = wo*vec(sload) + wt*vec(voidTy) + wa*vec(constant)
   let x := sload(a)
   Ctx[x] = wo*vec(sload) + wt*vec(voidTy) + wa*vec(Ctx[a])
   let mask := 0xffffffffffffffff
   Ctx[mask] = wo*vec(sload) + wt*vec(voidTy) + wa*vec(Ctx[a])
   (...)
   and(x, not(mask))
   wo*vec(and) + wt*vec(booleanTy) + wa *
   (wo*vec(not) + wt*(booleanTy) + wa(Ctx[mask]))
}
```

Figure 4. Whenever the vectorization process encounters a variable assignment, the assigned expression is vectorized and stored in the context. This vector representation is then used whenever the variable appears in the code. When the variable is referenced, its computed vector is substituted as the argument value.

4.4. Aggregated Yul Code Vectorization

The final vector representation of a given Yul program is obtained by summing the vectors of all statements within the main block of the code. The vector representation of each statement, in turn, is computed as the sum of the vectors of the expressions contained within it. In other words, to derive the final program vector, I traverse the Abstract Syntax Tree (AST) of the Yul code in a **depth-first search** (DFS) manner, aggregating the vectors of each expression encountered. Since statements can either be single expressions or combinations of multiple expressions (e.g., a *for-loop* statement consisting of multiple expressions, including loop conditions and nested expressions inside the loop body), the traversal ensures that all relevant expressions are considered.

However, certain expressions yield a **zero vector** as their direct result. This occurs in cases such as:

- **Variable definitions:** A variable declaration without an assigned value does not contribute to the aggregated vector.
- **Assignments:** As described in the previous subsection, assignments do not produce a direct vectorized representation. Instead, the vector of the assigned expression is computed and stored in memory (referred to as the **context map**) for future use, but the assignment operation itself results in a zero vector.
- **Function definitions:** Defining a function does not contribute directly to the final aggregated vector, as function definitions return a zero vector. However, the function's vector representation is computed and stored in the **context map**, ensuring that whenever the function is invoked, its computed vector is retrieved and incorporated into subsequent expressions.

The depth-first aggregation approach ensures that nested expressions are fully evaluated before their parent expressions, maintaining a structured and context-aware vectorization process. The AST traversal must be performed twice: the first pass collects all function definitions into the context map, while the second pass carries out the actual evaluation to compute the final vector.

A pseudo-code implementation of the final Yul code vectorization process is presented in Listing 3, illustrating how expressions and statements are recursively processed to construct the final vector representation.

Listing 3. Vectorizing Yul code with use of previously learned embeddings and with tracking the context (for variables). It is worth to mention that vectorizing should happen twice for each Yul contract: once to collect all function in the context and calculate them and the second one to get final vector.

```
function vectorize_arguments(ast):
    final_vector = []
    for arg in ast:
        if arg["nodeType"] eq "YulFunctionCall":
            final_vector += vectorize_function_call(arg)
        else if arg["nodeType"] eq "YulIdentifier":
            final_vector += context[arg["name"]] || embeddings["variable"]
        else if arg["nodeType"] eq "YulLiteral":
            final_vector += embeddings["constant"]
    return final_vector

function vectorize_function_call(ast):
    op_code = ast["functionName"]["name"]
    if op_code in EVM_OPCODES[op_code]:
        op_code_type = EVM_OPCODES[op_code]
        op_code_vec = embeddings[op_code] || embeddings["functioncall"]
    else:
        op_code_type = 'unknownTy'
        if op_code not in context["functions"]:
            op_code_vec = embeddings["functioncall"]
        else:
            if context["functions"][op_code]["calculated"]:
                op_code_vec = context["functions"][op_code]["vector"]
            else:
                op_code_vec = vectorize_ast(context["functions"][op_code]["code"])
                context["functions"][op_code]["calculated"] = true
                context["functions"][op_code]["vector"] = op_code_vec

    return op_code_vec * ow + embeddings[op_code_type] * tw + vectorize_arguments(ast['arguments']) *
        aw

def vectorize_ast(ast, ctx):
    nodeType = ast["nodeType"]

    match nodeType:
        case "YulObject":
            return vectorize_ast(ast["code"])
        case "YulCode":
            return vectorize_ast(ast["block"])
        case "YulBlock":
            final_vector = []
            for statement in ast["statements"]:
                final_vector += vectorize_ast(statement)
            return final_vector
        case "YulFunctionCall":
            return vectorize_function_call(ast)
        case "YulExpressionStatement":
            return vectorize_ast(ast["expression"])
        case "YulVariableDeclaration" | "YulAssignment":
            var_vector = vectorize_ast(ast["value"])
            for var in ast["variableNames"]:
                context[var["name"]] = var_vector
            return []
        case "YulFunctionDefinition":
            if ast["name"] not in context["functions"]:
                context["functions"][ast["name"]] = {"calculated": False, "vector": [], "code": ast["body"]}
            return []
        case "YulIf":
            return embeddings["if"] * ow + embeddings["booleanTy"] * tw + vectorize_ast(ast['condition']) *
                aw + vectorize_ast(ast['body'])
        case "YulForLoop":
            final_vector = []
            final_vector += embeddings["forloop')] * ow
            final_vector += embeddings["voidTy"] * tw
            final_vector += vectorize_ast(ast["pre"])
            final_vector += vectorize_ast(ast["condition"]) * aw
            final_vector += vectorize_ast(ast["post"])
            final_vector += vectorize_ast(ast["body"])
            return final_vector
        case "YulSwitch":
            final_vector = []
            final_vector += embeddings["switch"] * ow
            final_vector += embeddings["voidTy"] * tw
            final_vector += vectorize_ast(ast["expression"]) * aw
            for switch_case in ast["cases"]:
                final_vector += vectorize_ast(switch_case["body"])
            return final_vector
        case "YulBreak":
            return embeddings["break"] $*$ ow + embeddings["voidTy"] $*$ tw
        case "YulContinue":
            return embeddings["continue"] $*$ ow + embeddings["voidTy"] $*$ tw
        case "YulLeave":
            return embeddings["leave"] $*$ ow + embeddings["voidTy"] $*$ tw
        case "YulIdentifier":
            return context[ast["name"]] | | embeddings["variable"]
        case "YulLiteral":
            return embeddings["constant"]

for yul_code in yul_dataset:
    ast = parse_ast(yul_code)
    vectorize_ast(ast)
    vectorize_ast(ast)
```

5. Experimental Results

To validate the proposed approach, experiments were conducted on a dataset [13] comprising over 340,000 Yul files. All evaluations were performed on a machine equipped with a 10-core Apple M2 CPU and 32 GB of RAM. The evaluation process is divided into two main categories: (1) embeddings of fundamental Yul entities, which serve as the foundation for subsequent processing tasks, and (2) holistic vectorization of complete Yul contracts.

To the best of the author's knowledge, the vectorization of Yul contracts has not been explored in previous literature. As a result, there are no established benchmarks or directly comparable methods available for evaluation. While this limits the ability to perform standard comparative analysis, it also highlights the novelty and pioneering nature of the proposed approach. In light of this, a thorough internal evaluation was conducted to assess both performance and representational quality. The results aim to provide an initial baseline for future research in this area.

5.1. Yul Entities Embeddings

Basic embeddings of Yul elements were analyzed to demonstrate that the relationships between entities are effectively captured, and that the proposed method is capable of identifying the semantic characteristics of Yul instructions and their arguments. To facilitate the interpretation of results, entities were categorized into several functional groups: function arguments, data types, logical operations, arithmetic operations, memory/storage operations (for reading and writing data to memory or blockchain storage), and call operations (used to invoke functions from external contracts).

To visualize the relationships and distances between these entities, the 300-dimensional embedding vectors were projected into a two-dimensional space using Principal Component Analysis (PCA). The results of this dimensionality reduction are shown in Figure 5.

In Figure 5a, we observe that the three representative function argument types are clearly separated and equidistant from one another, indicating that the model distinguishes them well.

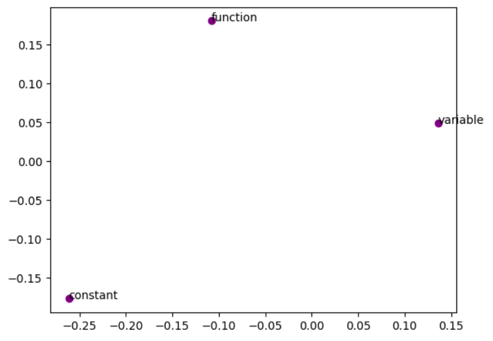
Figure 5b presents a similar scenario, particularly highlighting a significant distance between *numberTy* and *unknownTy*. The latter appears frequently in cases where the type is ambiguous or undefined - such as outputs from user-defined functions - explaining its divergence from the more precisely defined types.

In Figure 5c, a clear distinction between logical operations and argument types is evident. The logical operations are roughly evenly distributed around the argument types, suggesting that these operations interact with various types uniformly, reflecting the generality of logical constructs in the Yul language.

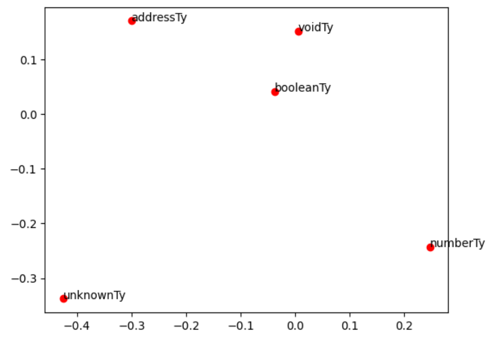
Figure 5d reveals a pronounced separation between argument types and arithmetic operations, as well as noticeable clustering among the operations themselves. A stronger association is observed between arithmetic operations and variable/function-related argument types, as opposed to constants. This aligns with common developer practices, where variables and function results are more frequently involved in arithmetic computations than constant values.

In Figure 5e, the store operations are evenly and distinctly distributed, indicating that the model recognizes their individual semantics. Notably, the proximity of the *mstore* operation to the *variable* argument type supports the intuitive understanding that memory storage is commonly used to hold variable values during execution.

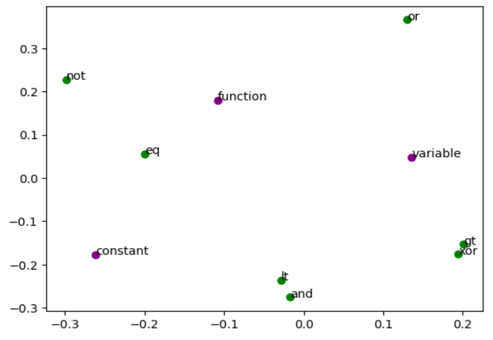
Lastly, Figure 5f compares logical, arithmetic, and call operations. Although distinct clusters for each category are not strongly pronounced, entities within the same category tend to be positioned relatively closer to one another. Moreover, the overall separation between different categories suggests that the embedding space reflects their functional distinctions to a reasonable extent.



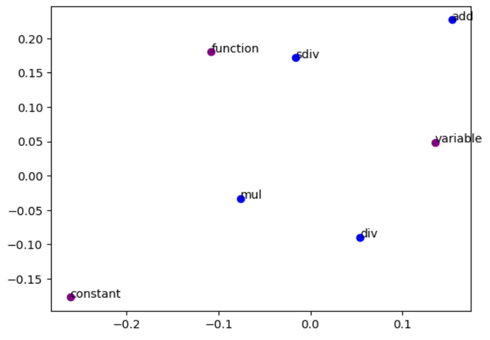
(a) Arguments



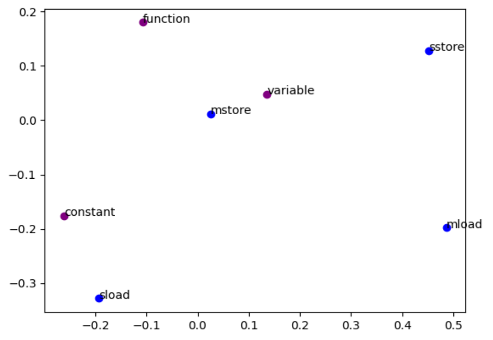
(b) Types



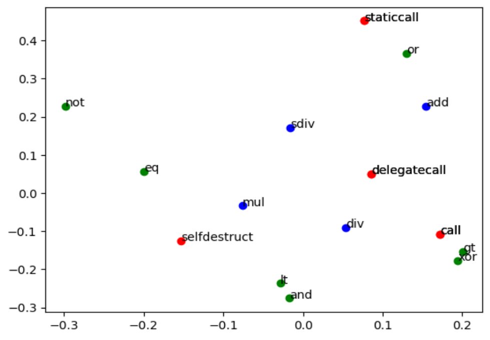
(c) Arguments vs Logical Operations



(d) Arguments vs Math Operations



(e) Arguments vs Memory Operations



(f) Logical vs Math vs Call Operations

Figure 5. Comparison of embeddings of various Yul entities categorized into several groups.

5.2. Yul Vector Distribution

This subsection presents a key contribution of the paper—the distribution of vectors generated from entire Yul programs using the method described in Section 4.4. The previously mentioned dataset, containing approximately 350,000 samples, was used. For the sake of clarity in presentation, a representative subset of these samples was selected.

A central goal of this method is to automatically capture similarities between contracts without human supervision. Paradoxically, this also makes it difficult to evaluate or interpret the quality of the resulting vector distributions using human judgment alone, as there is no clearly defined notion of an "ideal" distribution.

To address this challenge, two evaluation tests were designed:

1. **Automated Categorization of 2,000 Samples:** Embeddings were computed for the first 2,000 entries from the dataset and automatically assigned to one of five categories:
    - **Initial Code (red)** — Yul code produced as the constructor portion during Solidity compilation. Such code typically includes boilerplate logic for deploying contract bytecode onto the blockchain.
    - **Libraries (blue)** — Well-known and commonly used libraries, such as `Strings` or `Base64`.
    - **ERC-20 Contracts (green)** — Contracts implementing the ERC-20 standard for fungible tokens.
    - **ERC-721 Contracts (yellow)** — Contracts implementing the ERC-721 standard for non-fungible tokens.
    - **Other (purple)** — Contracts not falling into any of the above categories.

    Classification into the first two categories is highly reliable: initial code files typically have the suffix `"initial"` in their filenames, while library files often use the library name as a prefix. For ERC-20 and ERC-721 contracts, classification was based on the presence of relevant keywords (`ERC20`, `ERC721`) in the code. Although this heuristic is not perfect, it is sufficiently accurate for identifying general distributional trends.
2. **Manual Evaluation on Selected Contracts:** A small set of contracts was manually selected, with known categories based on prior knowledge. These samples were used to validate the embedding-based grouping from a qualitative standpoint.

Despite the inherent limitations of automated and heuristic-based categorization, this dual evaluation approach offers useful insights into the representational capacity of the vectorization method. As noted earlier, the goal is not necessarily for humans to interpret or label Yul programs but rather to enable the method itself to learn and represent latent similarities between them. In future work, a more robust indicator of quality could be derived by using these vectors as inputs to downstream AI tasks—for instance, optimization problems like phase-ordering in binary code compilation.

The results of Test 1 are presented in Figure 6. Clear clusters are observed for both initial code and library contracts, which are located near each other yet distinctly separated from other categories. ERC-20 contracts, though more widely dispersed along the X-axis, tend to concentrate along the lower region of the Y-axis. In contrast, ERC-721 contracts are mostly positioned above ERC-20 samples. The "Other" category does not form a well-defined cluster, which may be attributed to the limitations of automatic categorization. However, it is also reasonable to assume that contracts outside the predefined categories could still share similar characteristics with them.

Figure 7 shows the results of Test 2. Similar trends are evident: ERC-20 contracts are concentrated in the bottom-left region, while ERC-721 contracts gravitate toward the center-top area. Contracts from the "Other" category cluster in the mid-left region, a pattern also visible in Test 1. Initial codes appear tightly clustered in a distinct area, with libraries located nearby but slightly offset. In some cases, red and blue points overlap—these represent initial codes corresponding to library contracts.

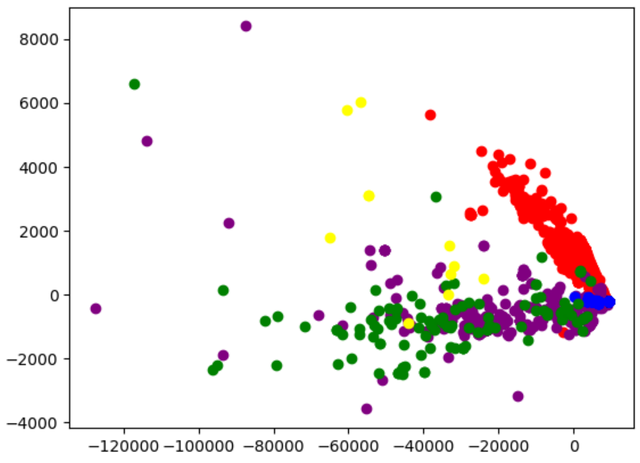


Figure 6. Distribution of vectors computed with yul2vec method. Categories: initial codes(red), libraries (blue), ERC-20/NT contracts (green), ERC-721/NFT contracts (yellow), others (purple).

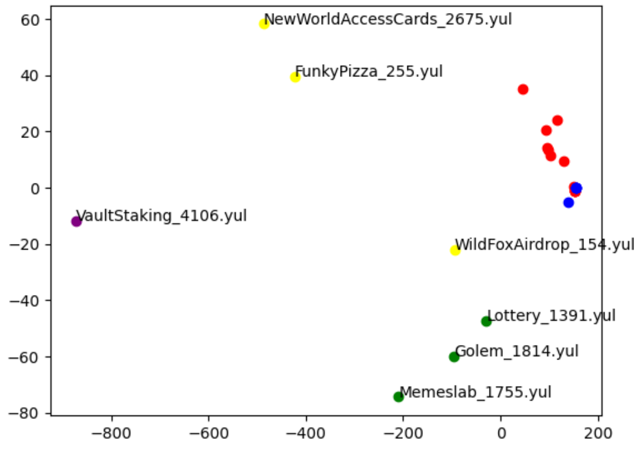


Figure 7. Distribution of vectors computed with yul2vec method for selected contracts of various categoties. Categories: initial codes(red), libraries (blue), ERC-20/NT contracts (green), ERC-721/NFT contracts (yellow), others (purple).

6. Summary and Future Work

Although the results are challenging to evaluate due to the absence of established benchmarks - and given the inherently abstract goal of enabling machines to capture latent semantic characteristics not easily interpretable by humans - they nevertheless appear promising. Despite these challenges, the experiments demonstrate that the vector representations are meaningfully clustered according to basic contract categories. This indicates that the proposed method is capable of capturing non-trivial patterns in Yul code, far beyond what would be expected from a random or purely syntactic representation.

These findings provide an encouraging outlook for the future application of this approach. In particular, the generated vectors show potential as high-quality input features for downstream neural network models. One especially compelling direction is the application of these embeddings to compiler optimization tasks, such as the phase-ordering problem in the Solidity compiler. By integrating these learned representations into optimization pipelines, it may become possible to automate and improve performance-tuning processes that currently rely heavily on expert heuristics.

Future work will focus on evaluating the embeddings in practical downstream tasks, exploring alternative aggregation strategies for program-level vectorization, and benchmarking against emerging methods as this area of research evolves.

Data Availability Statement: The data used for this research comes from the dataset published on HuggingFace: https://huggingface.co/datasets/KrzysztofFonal/yulcode. The code for generating embeddings can be found here: https://github.com/krzysiekfonal/solc-optimizer/blob/main/triplets_generator.py. The code for generating the final vectors from Yul contracts, using the vocabulary produced by the script above, is available as a Python notebook here: https://github.com/krzysiekfonal/solc-optimizer/blob/main/embedding_dataset_fa_generator.ipynb.

Acknowledgments: I would like to thank my friend Kamil Śliwak, a member of the Solidity compiler team, for sharing deep internal knowledge and offering many valuable tips.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Buterin, V. A next-generation smart contract and decentralized application platform. *white paper* **2014**.
2. Foundation, E. YUL documentation, 2018.
3. Muchnick, S.S. *Advanced compiler design and implementation*; Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 1998.
4. Almagor, L.; Cooper, K.D.; Grosul, A.; Harvey, T.J.; Reeves, S.W.; Subramanian, D.; Torczon, L.; Waterman, T. Finding effective compilation sequences. *ACM SIGPLAN Notices* **2004**, *39*, 231–239.
5. Cooper, K.D.; Schielke, P.J.; Subramanian, D. Optimizing for reduced code space using genetic algorithms. In Proceedings of the Proceedings of the ACM SIGPLAN 1999 workshop on Languages, compilers, and tools for embedded systems, 1999, pp. 1–9.
6. Cooper, K.D.; Subramanian, D.; Torczon, L. Adaptive optimizing compilers for the 21st century. *The Journal of Supercomputing* **2002**, *23*, 7–22.
7. Kulkarni, P.; Zhao, W.; Moon, H.; Cho, K.; Whalley, D.; Davidson, J.; Bailey, M.; Paek, Y.; Gallivan, K. Finding effective optimization phase sequences. *ACM SIGPLAN Notices* **2003**, *38*, 12–23.
8. Kulkarni, S.; Cavazos, J. Mitigating the compiler optimization phase-ordering problem using machine learning. *SIGPLAN Not.* **2012**, *47*, 147–162. https://doi.org/10.1145/2398857.2384628.
9. Jain, S.; Andaluri, Y.; VenkataKeerthy, S.; Upadrasta, R. POSET-RL: Phase ordering for Optimizing Size and Execution Time using Reinforcement Learning. In Proceedings of the 2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2022, pp. 121–131. https://doi.org/10.1109/ISPASS55109.2022.00012.
10. Huang, Q.; Haj-Ali, A.; Moses, W.; Xiang, J.; Stoica, I.; Asanovic, K.; Wawrzynek, J. AutoPhase: Compiler Phase-Ordering for HLS with Deep Reinforcement Learning. In Proceedings of the 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2019, pp. 308–308. https://doi.org/10.1109/FCCM.2019.00049.
11. Alon, U.; Zilberstein, M.; Levy, O.; Yahav, E. code2vec: learning distributed representations of code. *Proc. ACM Program. Lang.* **2019**, *3*. https://doi.org/10.1145/3290353.
12. VenkataKeerthy, S.; Aggarwal, R.; Jain, S.; Desarkar, M.S.; Upadrasta, R.; Srikant, Y.N. IR2VEC: LLVM IR Based Scalable Program Embeddings. *ACM Trans. Archit. Code Optim.* **2020**, *17*. https://doi.org/10.1145/3418463.
13. Fonal, K. Dataset of Yul Contracts to Support Solidity Compiler Research, 2025, [arXiv:cs.SE/2506.19153].
14. Allamanis, M.; Barr, E.T.; Bird, C.; Sutton, C. Suggesting accurate method and class names. In Proceedings of the Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, New York, NY, USA, 2015; ESEC/FSE 2015, p. 38–49. https://doi.org/10.1145/2786805.2786849.
15. Allamanis, M.; Brockschmidt, M.; Khademi, M. Learning to Represent Programs with Graphs. In Proceedings of the International Conference on Learning Representations, 2018.
16. Alon, U.; Zilberstein, M.; Levy, O.; Yahav, E. A general path-based representation for predicting program properties. *SIGPLAN Not.* **2018**, *53*, 404–419. https://doi.org/10.1145/3296979.3192412.
17. Brauckmann, A.; Goens, A.; Ertel, S.; Castrillon, J. Compiler-based graph representations for deep learning models of code. In Proceedings of the Proceedings of the 29th International Conference on Compiler Construction, New York, NY, USA, 2020; CC 2020, p. 201–211. https://doi.org/10.1145/3377555.3377894.
18. Mikolov, T.; Chen, K.; Corrado, G.; Dean, J. Efficient Estimation of Word Representations in Vector Space, 2013, [arXiv:cs.CL/1301.3781].
19. Pennington, J.; Socher, R.; Manning, C. GloVe: Global Vectors for Word Representation. In Proceedings of the Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP);

Moschitti, A.; Pang, B.; Daelemans, W., Eds., Doha, Qatar, 2014; pp. 1532–1543. https://doi.org/10.3115/v1/D14-1162.

20. Alon, U.; Brody, S.; Levy, O.; Yahav, E. code2seq: Generating Sequences from Structured Representations of Code. In Proceedings of the International Conference on Learning Representations, 2019.
21. Kanade, A.; Maniatis, P.; Balakrishnan, G.; Shi, K. Learning and Evaluating Contextual Embedding of Source Code, 2020, [arXiv:cs.SE/2001.00059].
22. Mou, L.; Li, G.; Zhang, L.; Wang, T.; Jin, Z. Convolutional neural networks over tree structures for programming language processing. In Proceedings of the Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence. AAAI Press, 2016, AAAI'16, p. 1287–1293.
23. Gupta, R.; Pal, S.; Kanade, A.; Shevade, S. DeepFix: fixing common C language errors by deep learning. In Proceedings of the Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence. AAAI Press, 2017, AAAI'17, p. 1345–1351.
24. Lin, Y.; Liu, Z.; Sun, M.; Liu, Y.; Zhu, X. Learning entity and relation embeddings for knowledge graph completion. In Proceedings of the Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence. AAAI Press, 2015, AAAI'15, p. 2181–2187.
25. Ji, G.; He, S.; Xu, L.; Liu, K.; Zhao, J. Knowledge Graph Embedding via Dynamic Mapping Matrix. In Proceedings of the Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers); Zong, C.; Strube, M., Eds., Beijing, China, 2015; pp. 687–696. https://doi.org/10.3115/v1/P15-1067.
26. Foundation, E. YUL documentation, 2014.
27. Foundation, E. YUL documentation, 2018.
28. Bordes, A.; Usunier, N.; Garcia-Duran, A.; Weston, J.; Yakhnenko, O. Translating Embeddings for Modeling Multi-relational Data. In Proceedings of the Advances in Neural Information Processing Systems; Burges, C.; Bottou, L.; Welling, M.; Ghahramani, Z.; Weinberger, K., Eds. Curran Associates, Inc., 2013, Vol. 26.
29. Han, X.; Cao, S.; Lv, X.; Lin, Y.; Liu, Z.; Sun, M.; Li, J. OpenKE: An Open Toolkit for Knowledge Embedding. In Proceedings of the Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations; Blanco, E.; Lu, W., Eds., Brussels, Belgium, 2018; pp. 139–144. https://doi.org/10.18653/v1/D18-2024.