Contents

List of Tables

List of Figures

Abstract

This thesis addresses the necessity for a new smart contract programming language within the Ethereum ecosystem, currently dominated by contracts written in Solidity. This dominance results in limitations such as monolingualism[1] and monoparadigmism[2]. Our research introduces Elixireum, a functional language featuring Elixir syntax and dynamic typing, designed to enhance the flexibility and creativity of decentralized application (dApp) developers. The Elixireum compiler was developed and tested for compatibility with the Ethereum Virtual Machine (EVM) and its ability to support ERC-20 and ERC-721 standards. A comprehensive test suite was employed to compare the gas consumption of Elixireum with that of Solidity. The results revealed that, although Elixireum is functional and compatible with Ethereum, it consumes more gas than Solidity. Key features, such as events and storage interaction, were implemented. However, limitations include high deployment gas consumption and the absence of external libraries and interaction with external smart contracts. Future work should focus on reducing deployment gas costs and implementing runtime mapping. Despite the higher gas consumption, Elixireum's ease of development and potential for earlier deployment could outweigh these costs, positioning it as a viable alternative in the Web3 space.

[1] Monolingualism - This term refers to the use of a single language by an individual or community, without the regular use of any additional languages.

[2] Monoparadigmism - This term describes the use of only one programming paradigm within a programming environment or by a programmer, excluding the use of multiple paradigms.

Chapter 1

Introduction

This chapter serves as an introduction to the Ethereum ecosystem, highlighting the popularity of the platform and its role in the crypto community. Section 1.1 provides a brief overview of Ethereum history and evolution. Section 1.2 underlines the demand for a new smart contract programming language and Section 1.3 states the goal of the thesis. Section 1.4 describes other thesis chapters.

1.1 Ethereum Overview

1. Ethereum: An Innovative Blockchain Platform

Ethereum, launched by Vitalik Buterin in July of 2015, is a major blockchain platform. It is supported by a large community of developers and cryptocurrency enthusiasts. To evaluate the importance and current popularity we can refer to CoinMarketCap[1]. Ethereum ranks 2nd after Bitcoin, having the market cap more than $380B and more than $10B daily trading value, indicating the significant role of ETH in the digital economy.

[1] CoinMarketCap - resources which provides analytics and prices for cryptocurrencies.

2. Blockchain Ecosystem and Core Features

Ethereum supports decentralized applications (dApps) and smart contracts. These contracts automatically manage transactions and agreements without needing a middleman. This system finds an application in various types of applications in areas like finance and supply chain management.

3. Programming Languages on Ethereum

Solidity is the most used programming language on Ethereum [1]. Vyper is the most popular alternative, however it shares a tiny part of the market. So, the scope of programming languages that can be used to develop decentralized applications is limited.

4. Importance of Writing Effective Smart Contracts

Using smart contracts on Ethereum costs money, known as "gas fees", which user i.e. transaction sender pays for each action a contract performs. These costs make it important to write efficient smart contracts to reduce expenses and improve performance.

5. The Rise of Layer 2 Rollups

With the increasing popularity and widespread adoption of Ethereum, the price of its native coin, ETH, has surged significantly. As the cost of ETH rises, even simple transactions like transferring coins can become too expensive, sometimes costing up to $100 per transaction. Such high fees restrict the practical usability of the blockchain. Consequently, rollup blockchains have emerged to address these challenges. The concept behind rollup solutions can be described

as a cheap blockchain (L2) on top of an expensive blockchain (L1). An L2 chain performs the execution of hundreds of transactions and then stores the resulting state changes within a single transaction on an L1. The fact that all the changes eventually are stored in an L1 blockchain enables an L2 chain to retain all L1 blockchain characteristics such as immutability and decentralization, while significantly reducing gas costs and increasing throughput. It allows developers to focus more on features and security, changing how dApps are made and used.

1.2 Demand for a New Language

DApp developers mostly use Solidity programming language. This language is imperative in nature. Monolingualism and monoparadigmism limit developers' flexibility and creativity. Additionally, Solidity has no support for certain features such as floating-point numbers, which are crucial for handling decimal values in calculations, thereby restricting its utility for a range of applications. Moreover, the high entry threshold associated with this language can deter newcomers from entering the field. It often requires a substantial investment of time and effort to become proficient, which can slow down the development process.

Given these constraints, there is a significant opportunity for a new language. A functional alternative, particularly one that supports dynamic typing and builds upon the foundations of a widely recognized programming language, could dramatically enhance the accessibility and efficiency of dApp development. Such a language would not only make it easier for new developers to adopt but also open up new possibilities for innovation in the Web3 space. This potential makes the development of a new language a promising avenue for expanding the capabilities and reach of the programming environment of Ethereum.

1.3 Goal of the Thesis

1. Research Aims and Hypotheses

Our goal is to create a smart contract language with Elixir syntax and dynamic types – Elixireum. As a result, language should be expressive at least to be able to implement ERC-20[2] and ERC-721[3] tokens. We plan to add such features as immutability, macros support and pattern matching to make this language functional. Also, we expect that Elixireum will have the same or less gas consumption in comparison with Solidity.

2. Expected Results

- **Working Compiler for Elixireum:** Development of a compiler capable of translating Elixireum-specific language into EVM bytecode.
- **Deployed ERC-20 Token:** Deployed ERC-20 token written on Elixireum as a showcase that Elixireum has fundamental functionality needed for smart contract development.
- **Compared Gas Consumption:** Analysis of gas usage for Elixireum contracts and comparative evaluation with Solidity.

1.4 Thesis overview

This section overviews the thesis. Chapter 2 Literature Review contains an introduction to Ethereum and a review of the existing related literature. Chapter

[2]https://eips.ethereum.org/EIPS/eip-20
[3]https://eips.ethereum.org/EIPS/eip-721

3 Methodology includes a theoretical description of approaches which was used during the implementation phase. Chapter 4 Implementation provides a detailed description of the Elixireum implementation. Chapter 5 Results and Discussion shows the result we achieved and compares the Elixireum with other languages. Chapter 6 Conclusion concludes the research, and states the limitations and future work scope.

Chapter 2

Literature Review

This chapter serves as an introduction to the Ethereum ecosystem and offers an overview of the relevant literature.

Section 2.1 introduces the Ethereum Virtual Machine (EVM) and explores its gas consumption mechanism. Following that, Section 2.2 examines of the existing languages compatible with the EVM. Section 2.3 presents an overview of the Elixir programming language. To conclude, Section 2.4 concludes this chapter with key decisions and findings from the literature.

2.1 Ethereum Virtual Machine

According to [2], Ethereum is a decentralized system, and at its core, the EVM is an essential component of the Ethereum blockchain. The EVM serves as the computational heart of the Ethereum network, enabling the execution of smart contracts and providing the foundation for a wide array of decentralized applications.

EVM has a significant security mechanism – gas limit. Each operation and

computation on the EVM consumes a certain amount of gas – a measure of computational work. The gas limit is a cap set by the user or the entity initiating a contract execution, and it represents the maximum amount of gas they are willing to spend for that operation.

The gas limit, the stack-based architecture of the EVM, and the broad set of available operations allow it to execute these smart contracts efficiently and securely for miners. Also, Buterin [2] defines EVM as a Turing-complete machine. This level of computational flexibility empowers developers to create sophisticated programs.

2.2 Existing EVM languages

Official Ethereum documentation [3] enumerates only four languages that compile to EVM: Solidity, Vyper, Yul, and Fe. Also, other attempts to create a language for EVM are known[1], but only officially documented languages will be considered in this paper.

Among the available languages for EVM, Solidity stands out as the most widely adopted and utilized one [1], [4]. This dominance is further emphasized by data presented in [2], which reveals that there are approximately 800 times more Solidity files on Github than Vyper files. Consequently, most existing smart contracts on the Ethereum blockchain are written in Solidity. On the one hand, the concentration of development and resources around a single language like Solidity can be viewed positively. It fosters a strong sense of community cohesion and ensures that the majority of developers are focused on refining and improving a

[1] A curated collection of resources on smart contract programming languages
[2] Analyzing Solidity and Vyper for Smart Contracts Programming

single language, potentially enhancing its robustness and feature set. However, this dominance also brings significant challenges. The high prevalence of Solidity in the EVM ecosystem means that if a vulnerability or security flaw arises in the Solidity language, it affects a substantial portion of the smart contracts on the network. This centralized risk can be a double-edged sword, as a single point of failure in Solidity could have far-reaching consequences for the entire Ethereum blockchain. Furthermore, the Ethereum community is not without its criticisms of Solidity. Community members have voiced their concerns and challenges related to the language [4]. These concerns range from the complexity of the language to its lack of certain features, which may hinder the development of robust and secure smart contracts. So, decentralization and availability of choice are essential, especially for a decentralized ecosystem like blockchain.

1. Solidity

Solidity is a high-level, object-oriented programming, statically typed language explicitly designed for the EVM. Its syntax is similar to C++, Python, and Javascript [5]. It supports multiple inheritance, complex user-defined types, libraries, and some other features [6].

2. Vyper

Vyper contrasts significantly with Solidity. It is based on Python and extends it to suit smart contract development. Vyper aims for security and simplicity in terms of compiler implementation itself and code readability to be auditable more easily. For instance, it lacks a list of features considered harmful for code readability, making the language more error-prone [7].

2.3 The Elixir language

Elixir [8] is a modern programming language that operates on the Erlang Virtual Machine. With its support for immutable variables, Elixir dramatically enhances the clarity and simplicity of smart contract code. This immutability ensures that once data are defined, they cannot be altered, promoting predictability and security in the context of smart contract development.

A standout feature of Elixir is its metaprogramming capabilities. This functionality empowers developers to create domain-specific languages that are tailor-made for the intricacies of smart contract logic. This capability is invaluable as it allows developers to express contract logic in both highly readable and closely aligned with the specific problem domain way. By employing metaprogramming, smart contract development will become more straightforward and accessible to a broader spectrum of developers.

As attested by O'Grady [9], Elixir stands strong among the top 50 programming languages. Remarkably, it surpasses even the popularity of Solidity, making it an attractive choice for Ethereum developers and thereby contributing to the expansion of the Ethereum community.

In summation, our decision to adopt Elixir as the language for our compiler and as the foundational framework for the Elixireum language is bolstered by its immutability, metaprogramming capabilities, and prominence in the software development landscape. This choice ensures the simplicity and security of smart contract development and the growth and diversification of the Ethereum ecosystem.

2.4 Conclusion

In conclusion, this chapter has emphasized two significant matters. Firstly, the overwhelming dominance of Solidity has brought substantial challenges to the Ethereum ecosystem. Secondly, the absence of functional languages for smart contract development has limited the utilization of functional paradigm advantages. Consequently, we have decided to develop Elixireum, a functional programming language built on Elixir, explicitly tailored for smart contract development.

Chapter 3

Methodology

Chapter overview. Section 3.3 states metrics we used to evaluate the Elixireum language.

3.1 Compiler Construction

Compiler construction is a sophisticated field in computer science that involves the creation of software that translates source code written in a high-level programming language into a lower-level language, often machine code. This process is crucial as it serves as a bridge between high-level programming languages and the machine code that the Ethereum Virtual Machine (EVM) understands. The process can be divided into several distinct phases, each with its specialized function:

1. Lexical Analysis

According to Waite and Goos [10, pp. 135–148], the initial step in the compilation process is lexical analysis, also known as tokenization. During this phase,

the source code is broken down into smaller units called tokens, which include keywords, identifiers, operators, and constants. Lexical analyzers often use regular expressions to recognize these tokens, simplifying the source code into a stream of tokens for further processing.

2. Syntax Analysis

Following tokenization, the next phase is syntax analysis, or parsing. During this phase, the compiler checks the arrangement and structure of tokens to create a structured representation, often in the form of a syntax tree or abstract syntax tree (AST). This representation helps ensure that the code conforms to the grammar of the language rules and can be further processed [10, pp. 149–182].

3. Semantic Analysis

Semantic analysis involves ensuring that the parsed code makes logical sense according to the language's rules. This includes type checking, where the compiler verifies that each operation is compatible with its operands. The compiler also uses a symbol table to keep track of variable names, their types, and other relevant information to ensure correct interpretation and context.

4. Intermediate Code Generation

While not always mandatory, the use of an intermediate representation can significantly enhance the efficiency and capabilities of a compiler. These representations help enhance code quality, optimize performance, and prepare the code for final code generation. This intermediate code is typically platform-independent, serving as a bridge between the high-level source code and the ma-

chine code. In the context of the EVM, intermediate representations like Yul and Yul+ are used for optimizing code.

5. Optimization

The optimization phase refines the intermediate code to improve efficiency and performance. This can include eliminating redundant calculations, optimizing loops, and performing target-dependent optimizations based on the architecture and capabilities of the target machine. This stage is particularly important in scenarios where verification, security, and performance optimizations are key concerns.

6. Code Generation

The final stage of the compilation process is code generation [10, pp. 253–281]. During this phase, the compiler generates the actual machine code that the target platform can execute. The generated code is specific to the hardware architecture and represents a translation of the original high-level code into instructions that the target can understand and execute. Sometimes, the output is in the form of assembly language, which is then further translated into machine code by an assembler.

7. Conclusion

In the context of Elixireum, these compiler construction principles must be adapted to suit the specific needs of the EVM. This includes considerations for smart contract compilation, optimization for gas efficiency, and adhering to the deterministic execution model of the EVM. Understanding these compiler con-

struction phases is essential for Elixireum development, as each stage plays a critical role in transforming high-level Elixireum code into efficient, secure, and EVM-compatible bytecode.

3.2 Ethereum Virtual Machine - A Deeper Insight

EVM is a core component of the Ethereum blockchain, pivotal in maintaining its decentralized and secure nature. It serves as an execution environment for smart contracts, enabling the remarkable functionality that Ethereum offers. Understanding the EVM is crucial for developing Elixireum, as it provides the foundational framework within which this new language operates.

1. Execution Environment

    - Isolated System: The EVM operates as an isolated environment, completely sandboxed from the Ethereum network. This isolation ensures that code execution within the EVM does not directly affect the network or the filesystem of the host computer.
    - Deterministic Execution: The EVM is designed to be deterministic. This means that executing the same smart contract with the same input will always produce the same output across all nodes in the Ethereum network. This characteristic is essential for maintaining consistency and agreement (consensus) across the decentralized network.

2. Role of EVM in Smart Contracts

- Smart Contract Lifecycle: Developers write smart contracts in high-level languages like Solidity or Vyper. These contracts are then compiled into EVM bytecode. When a contract is deployed on the Ethereum blockchain, it's executed by the EVM to ensure it behaves as expected.
- Gas Mechanism: The EVM uses a gas mechanism to measure and limit the resources (computation and storage) a contract can use. Each operation in the EVM has a gas cost associated with it, preventing inefficient or malicious code from overloading the network.

3. State Machine

   - State Transitions: The EVM can be viewed as a state machine, where transactions transition the Ethereum blockchain from one state to another. Smart contracts, through their executions, modify the state stored in the Ethereum blockchain.
   - EVM Bytecode: The EVM interprets a series of bytes (bytecode) which instruct the machine on the operations to perform. This bytecode is the compiled version of high-level contract code.

4. Storage and Memory

   - Storage: The EVM has a long-term storage model, where data is stored persistently on the blockchain. This storage is expensive but necessary for maintaining the state across transactions.
   - Memory: The EVM also includes a volatile memory space, cleared after each transaction, used for temporary storage during contract execution.

5. Challenges and Considerations for Elixireum

    - Gas Optimization: Given the cost implications of EVM operations, Elixireum must focus on gas optimization during the compilation process.
    - Security and Determinism: Security vulnerabilities in smart contracts can be costly. Hence, Elixireum needs to ensure robust security measures. Additionally, the deterministic nature of the EVM requires that Elixireum-generated bytecode behaves consistently across all executions.

3.3 Testing and Benchmarking

To optimize Elixireum development, we decided to write a test suite to cover the implemented functionality. After each feature implementation or bug fix, we ran the test suite to ensure that no previous functionality was broken. The test suite includes a set of smart contracts that collectively utilize almost all the implemented features, as well as a set of corresponding test Python files.

Within each Python test file, a test module is defined. Its constructor deploys the smart contracts, and unit tests are written within test functions to ensure that the smart contracts behave as expected. All smart contract interactions are implemented using the Web3.py library, and the utilized test framework is pytest. As the test EVM environment, we used Anvil[1], a local JSON-RPC node.

For comparison purposes, we also wrote a similar test suite in Python, which deploys the Elixireum contract and its analogue in Solidity. We performed sev-

[1] `https://github.com/foundry-rs/foundry/tree/master/crates/anvil#anvil`

eral runs of each function, measuring the gas consumption of each function and retrieving this information from the transaction receipts provided by the node. We created two modules for comparing ERC-20 and ERC-721 pairs of contracts.

Chapter 4

Design and Implementation

This chapter outlines the principal elements of the design of Elixireum compiler. It overviews the technology stack employed in the development process and underscores the pivotal decisions that shaped the architecture of the compiler.

4.1 Stack overview

This section rationalize the technology stack used in the project: Elixir, Yul, Solidity compiler.

1. Choice of Elixir

Here is the rationalization behind the choice of Elixir:

- One of the most popular open source blockchain explorers, Blockscout[1], is written in Elixir. From this we can say that Elixir is well heard of in Web3 community.

[1] https://www.blockscout.com – Official Blockscout website

- Our team possesses substantial expertise in this programming language.
- Elixir is open source so we can easily reuse its machinery for tokenization and parsing Elixireum.

2. *Choice of Yul*

- Initially, our strategy was to compile Elixireum directly into Ethereum Virtual Machine (EVM) bytecode. Subsequently, we pivoted towards an intermediate representation (IR), selecting Yul as IR aligned with our requirements. Advantages from this decision are as follows:
  - Compatibility with all EVM versions, eliminating the need for version-specific concerns.
  - Avoidance of reimplementing primitives such as function calls, variable and stack management, thereby freeing up resources to enhance the feature set of the language.
  - Reduced gas consumption facilitated by the optimizer of Solidity compiler.

  Despite these considerations, the choice comes with limitations, notably:

  - Dependency on the Solidity compiler for Elixireum.
  - The unresolved bug[2] in the Solidity compiler that appears for large smart contracts written in Elixireum when compiled with the optimizer disabled. The bug results in an inability to operate with a large number of variables or deep stack depths.

[2] https://github.com/ethereum/solidity/issues?q=is:issue+is:open+StackTooDeepError

No significant downsides compared to direct compilation to EVM bytecode, since bytecode can be used in Yul directly.

- We also considered mapping BEAM (Erlang VM) bytecode to EVM bytecode. However, this option presented significant challenges due to the fundamental architectural and philosophical differences between the two virtual machines. BEAM is specifically tailored for environments that demand high concurrency, distribution, and fault tolerance, and it includes a set of opcodes optimized for these conditions. In contrast, the EVM is designed to run smart contracts within blockchain ecosystems, prioritizing deterministic execution and gas metering to protect against spam and enhance network security. BEAM incorporates specialized instructions for concurrency and distributed processing that do not directly translate to the EVM, because of these distinct focuses.

4.2 Technical details

1. Memory organization

In our language implementation, we decided to store all runtime variables in the volatile memory of the Ethereum Virtual Machine. This decision is justified by the low gas cost associated with memory operations and the simplicity of implementation. We considered two alternatives: transient storage, which is more expensive than memory, and the stack, which involves more complex management.

In the generated Yul code, a variable is defined as a pointer to the memory location where its value is stored. At this pointer, the first byte is reserved for the

type number, and the subsequent bytes contain the value itself for simple cases. For more complex cases, such as lists and tuples, the first byte still indicates the type, followed by a word[3] that specifies the count of elements, and then the elements themselves, each stored as an inline variable (type + value). For strings or byte arrays, the first byte indicates the type (1 for strings, 102 for byte arrays), the next word specifies the byte count, followed by the actual bytes.

Since we developed a functional language, one of its fundamental principles is immutability. This means that each assignment operation creates a new variable. In practice, we "forget" the previous pointer, assign a new pointer, and place the variable in a new location. Each new variable should be stored sequentially after the previous one. To achieve this, we define the offset$ variable in the generated Yul code. This variable holds the memory address where the next variable will be stored. As a workaround for corner cases, we update this variable using the msize() call, assigning the current memory size rounded to one word to the offset$ variable.

In Elixireum syntax, there are no type specifiers for variables. To handle dynamic types, we established several principles upon which our type system is built:

- Return values of public functions must strictly match the specified types; otherwise, a revert[4] operation will be triggered.
- For literals, the type is defined at compile time.
- The type is stored in the first byte of the variable.

[3] in this context word is 32 bytes

[4] opcode which stops the current execution and throw an error. All the state changes are reverted

- We defined a standard cast function that developers can use in Elixireum code.

  The following code snippet demonstrates the usage of the cast function:

```
@spec decimals() :: Types.UInt8.t()
def decimals() do
  Utils.cast(18, Types.UInt8)
end
```

Listing 4.1: Example of cast function usage

The following Elixireum code illustrates a simple case:

```
a = 1
b = 2
```

Listing 4.2: Elixireum code for simple case

This Elixireum code is translated into Yul code as follows:

```
let offset$ := msize()      // define offset$, set it to current memory
    size
let a$ := offset$           // define variable as the memory pointer
mstore8(offset$, 67)        // 67 is int256 type, store it as one byte
offset$ := add(offset$, 1)  // increase current offset by 1 byte, which
    is used by type on the line above
mstore(offset$, 1)          // store the value itself
offset$ := add(offset$, 32) // int256 value takes 32 bytes

let b$ := offset$           // define variable as the memory pointer
mstore8(offset$, 67)        // 67 is uin256 type
offset$ := add(offset$, 1)  // increase current offset by 1 byte, which
    is used by type on the line above
mstore(offset$, 2)          // store the value itself
offset$ := add(offset$, 32) // int256 value takes 32 bytes
```

Listing 4.3: Generated yul code for simple case

In this Yul code, offset$ is initialized to the current memory size. Each variable (a$ and b$) is defined as a memory pointer. The type of each variable is stored using mstore8, and the actual value is stored using mstore. The offset is incremented appropriately to account for the type byte and the 32-byte value for each variable.

The following table shows a snippet of the EVM memory for this simple case:

Addresses	0x00	0x01	0x02	...	0x21	0x22	0x23	0x24	...	0x42
Values	0x43	0x00	0x00	...	0x01	0x43	0x00	0x00	...	0x01

TABLE I
EVM Memory snippet for simple case

An example of handling strings in Elixireum is as follows:

```
1  a = "abc" # hex representation of this utf-8 string is 0x616263
```

Listing 4.4: Elixireum code for string case

The corresponding Yul code generated for this string case is:

```
1  let offset$ := msize()      // define offset$, set it to current memory
       size
2  let a$ := offset$           // define variable as the memory pointer
3  mstore8(offset$, 1)         // store the type, 1 is the string type
4  offset$ := add(offset$, 1)  // increase current offset by 1 byte, which
       is used by type on the line above
5  mstore(offset$, 3)          // store the byte size of the string
6  offset$ := add(offset$, 32) // increase current memory pointer by 32
       bytes, since the size of the string is 256 bit lenght
7  mstore(offset$, 0
       x6162630000000000000000000000000000000000000000000000000000000000)
       \\ store the string itself
8  offset$ := add(offset$, 3)  // string has length only 3 bytes
```

Listing 4.5: Generated yul code for string case

In this Yul code, the variable a$ is defined as a memory pointer. The type of the variable is stored using mstore8, followed by storing the byte size of the string using mstore. The offset is incremented by the size of the string (3 bytes) after storing the string itself.

The following table illustrates a snippet of the EVM memory for this string case:

Addresses	0x00	0x01	0x02	...	0x20	0x21	0x22	0x23
Values	0x01	0x00	0x00	...	0x03	0x61	0x62	0x63

TABLE II
EVM Memory snippet for string case

An example of handling lists in Elixireum is as follows:

```
a = [1, true, "abc"]
```

Listing 4.6: Elixireum code for list case

```
let offset$ := msize()

// define int256 literal
let var0$ := offset$
...
// define bool literal
let bool_var1$ := offset$
...
// define string literal
let str2$ := offset$
...

// define the list itself
let a$ := offset$ // define variable as the memory pointer
mstore8(offset$, 103) // store the list type
offset$ := add(offset$, 1)
mstore(offset$, 3) // store the length of the list
```

```
18  offset$ := add(offset$, 32) // increase current memory pointer by 32
        bytes, since the size of the list is 256 bit lenght
19  let ignored_
20  ignored_, offset$ := copy_from_pointer_to$(var0$, offset$) // copy
        int256 variable to the start of the list & update offset$
21  ignored_, offset$ := copy_from_pointer_to$(bool_var1$, offset$) // copy
        bool variable to the list & update offset$
22  ignored_, offset$ := copy_from_pointer_to$(str2$, offset$) // copy
        string variable to the list & update offset$
```

Listing 4.7: Generated yul code for string case

In this Yul code the variables var0$, bool_var1$, and str2$ are defined for the integer, boolean, and string literals, respectively. Then list a$ is defined. The length of the list is stored using mstore, and the actual values of the list elements are copied into memory using the copy_from_pointer_to$ function, which updates the offset accordingly.

The following table illustrates a snippet of the EVM memory for this list case:

Addresses	...	0x47	0x48	...	0x67	0x68	0x69	...	0x88	0x89
Values	...	0x67	0x00	...	0x03	0x43	0x00	...	0x01	0x02
Addresses	0x8a	0x8b	0x8c	...	0xab	0xac	0xad	0xae	...	...
Values	0x01	0x01	0x00	...	0x03	0x61	0x62	0x63	...	...

TABLE III
EVM Memory snippet for list case

2. *Key features*

1. **Events**[5]

[5] https://docs.soliditylang.org/en/v0.8.25/abi-spec.html#events

Events are essential for highlighting storage changes or other significant events in the lifecycle of a smart contract. They can be easily retrieved from the blockchain after being emitted. Events can be defined and emitted in Elixireum as follows:

```
1  @test name: "Test", indexed_arguments: [addr: Types.Address.t()],
       data_arguments: [value: Types.UInt256.t()]
2    ...
3  addr = ~ADDRESS(0x0000000000000000000000000000000000000000)
4  value = 100
5  Event.emit(@test, [addr: addr value: value])
```

Listing 4.8: Events

2. **Storage**

Each address in Ethereum has its own associated persistent storage, which functions as a key-value store with 32-byte keys and values. This storage is crucial for maintaining a global state. In Elixireum, storage variables can be specified as a module argument using the @ symbol. The type of the storage variable can be any available type, a mapping, or an inner mapping of any depth.

```
1    @balances type: %{Types.Address.t() => Types.UInt256.t()}
2    @totalSupply type: Types.UInt256.t()
3    ...
4    Storage.store(@totalSupply, 1_000_000) # store the value to the
       storage
5    supple = Storage.get(@totalSupply)     # get the value from the
       storage
6
7    addr_1 = ... # define some address
8    Storage.store(@balances[addr_1], 1_000)  # update the @balances
       mapping for addr_1
```

```
9     balance = Storage.get(@balances[addr_1]) # get value for addr_1
          from @balances
```

Listing 4.9: Storage

3. **Constructor**

   The constructor is called once, during the deployment of the smart contract. It is used to initialize the contract, set the owner, or define any other storage variables. Essentially, it executes any necessary operations upon contract creation.

```
1     @spec constructor(Types.String.t(), Types.String.t()) :: nil
2     def constructor(name, symbol) do
3       Storage.store(@name, name)
4       Storage.store(@symbol, symbol)
5       Storage.store(@owner, Blockchain.caller())
6     end
```

Listing 4.10: Constructor

4. **ABI for Public Functions**

   The compiler enforces the specification of all public functions through compile-time errors. After compilation, all defined specifications are converted to the ABI (Application Binary Interface) JSON, which is necessary for correctly calling the smart contract after deployment.

```
1   @spec a(Types.Int256.t()) :: Types.String.t()
2   def a(_int) do
3     "return string"
4   end
```

Listing 4.11: @spec

Resulting JSON ABI:

```
    [
      {
          "name": "a",
          "type": "function",
          "outputs": [
              {
                  "name": "output",
                  "type": "string",
                  "internal_type": "string"
              }
          ],
          "inputs": [
              {
                  "name": "_int",
                  "type": "int256",
                  "internal_type": "int256"
              }
          ],
          "stateMutability": "view"
      }
  ]
```

Listing 4.12: ABI

5. **Reverts with UTF-8 Error String**

Developers can revert the execution of a smart contract, optionally specify an error string. The revert reason for the transaction will be a hex-encoded UTF-8 string.

```
    raise "ERC20InvalidSender(address(0))"
```

Listing 4.13: Reverts

4.3 Modules overview

This section provides a detailed overview of the main modules of the compiler. The modules are listed in the order corresponding to the contract life cycle i.e., deployment and then calls.

1. Yul compiler

The Yul compiler calls the Solidity compiler to generate bytecode, given the path to the standard JSON input file[6]. For this purpose, it looks for the Solidity compiler that is capable of compiling Yul to bytecode. If there is no Solidity compiler already downloaded, the Yul compiler module downloads platform specific Solidity compiler. Then, it returns standard JSON output[7].

[6] https://docs.soliditylang.org/en/v0.8.25/using-the-compiler.html#input-description

[7] https://docs.soliditylang.org/en/v0.8.25/using-the-compiler.html#output-description

2. Compiler

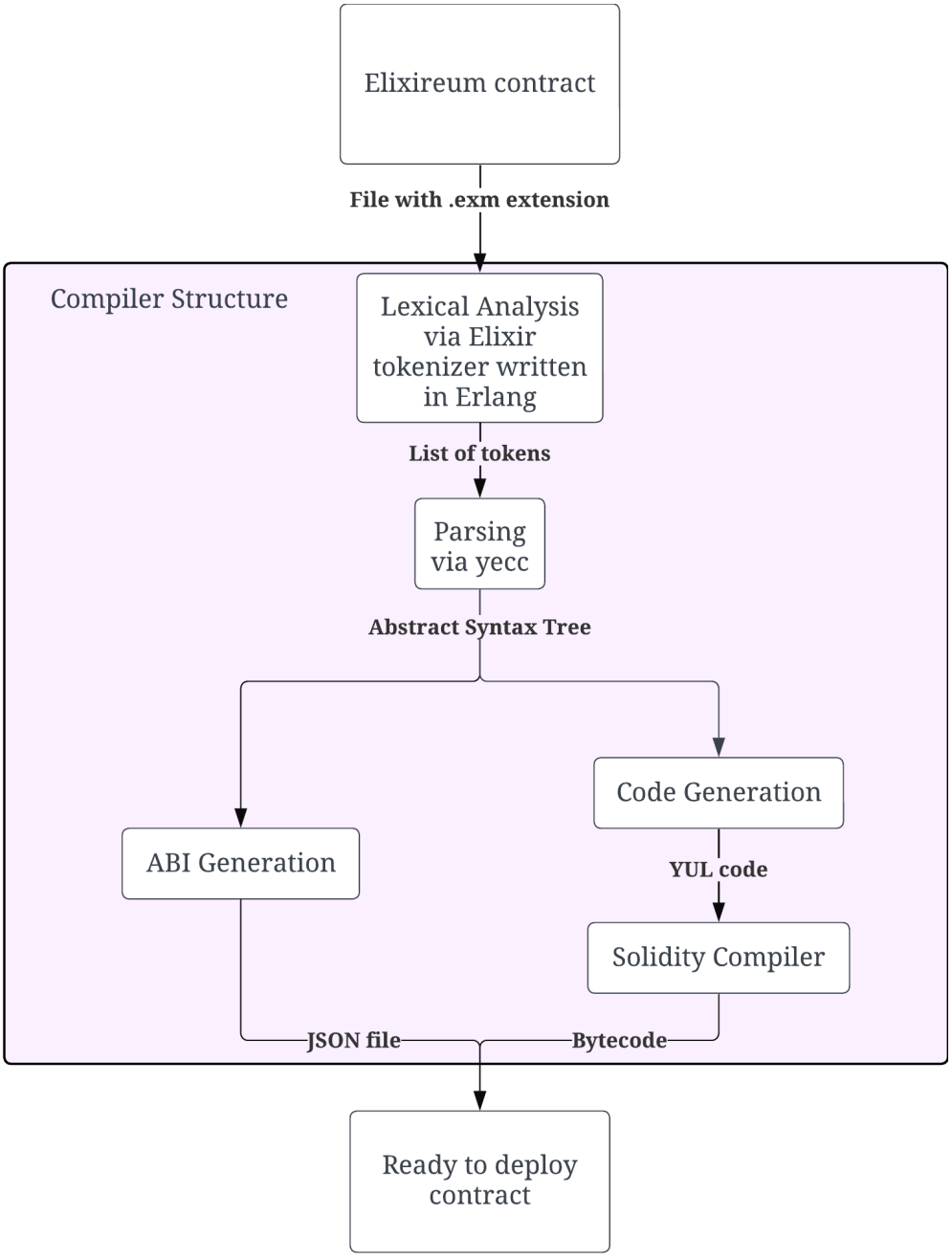

Fig. 1. Elixireum compiler architecture

Here is the pipeline of actions performed by the main Compiler module, which can be seen visually in Fig. 1, each pipeline step is performed only if all the previous steps are successful.

- Reading the source code from the file system.
- Running the Elixir tokenizer and parser on the source code.
- Performing the first pass of the Elixir AST in pre-order to gather information about defined functions, storage variables, and events, and packing this data into the following structure:

```
1       @type t :: %Contract{
2           functions: Functions.t(),
3           name: String.t(),
4           private_functions: Functions.t(),
5           variables: %{atom() => Variable.t()},
6           events: %{atom() => Event.t()},
7           aliases: %{atom() => list()}
8         }
```

Listing 4.14: Contract structure

  At this step, the compiler checks that public defined functions have corresponding typespecs[8], storage variables have types and events have all required fields.

- Perform the second traversal of the Elixir AST in post-order to generate Yul code.

  Specifically, this step generates a special function called a constructor. This function is executed at the deployment stage, and it takes its arguments not from calldata, but from the code. The Yul code that decodes arguments is generated using the calldata decoding module.

  Then, the compiler generates a function selector. The function selector is used to identify which function is being called based on the first four

[8]Typespec is the type specification of the function used in Elixir

bytes of the call data. The first four bytes of the call data should be equal to the first four bytes of the Keccak256 hash of the string in the following format: `function_name(1st_arg_type_name,2nd_arg_type_name,...)` constructed for the calling function, this is called method ID. For example, if the function has the following signature `transfer(address to, uint256 value)` Keccak256 is performed on `transfer(address,uint256)` The result of Keccak 256 is equal to 0xa9059cbb2ab0... So, the method ID for the transfer function is 0xa9059cbb. The function selector is simply a switch case expression where each public defined function corresponds to a case with its method ID. For each case statement, the compiler generates Yul code, responsible for function arguments decoding from call data, using the call data decoding module. Then the function call is generated as if it was a private function. Then, the compiler generates the code that encodes the value returned by the function into the return format using the return data encoding module.

Then, the compiler generates Yul code for all user-defined functions, both public and private, since the difference between public and private functions is factored out into the function selector.

Then, the compiler generates Yul code for standard functions used in user-defined functions. Standard functions are represented by the following structure:

```
1       @type t :: %StdFunction{
2           deps: %{atom() => t()},
3           yul: String.t()
4         }
```

Listing 4.15: Standard function structure

The field deps represents other standard functions used by this standard function. This mechanism allows the definition of standard functions on demand, thus reducing the size of the Yul code and lowering the cost of deployment. Here is the mechanism that resolves dependencies:

```
1   defp generate_std_functions(used_std_functions, definitions \\ %
        {}) do
2     used_std_functions |>
3     Enum.reduce(definitions, fn {function_name,
4                                  %StdFunction{yul: yul, deps: deps}
                                       },
5                                  definitions_acc ->
6       {_, not_defined_deps} = Map.split(deps, Map.keys(
            definitions_acc))
7
8       generate_std_functions(not_defined_deps, Map.put_new(
            definitions_acc, function_name, yul))
9     end)
10  end
```

Listing 4.16: Standard function dependencies

- Then, using the output of the first pass, the compiler generates ABI via ABI generator module.

3. ABI generator

The ABI generator takes the contract information collected by the compiler in the form of the contract structure 4.14 and generates a JSON ABI using the functions and events fields of the contract structure.

4. Calldata decoding

The call data decoding module recursively generates code for parsing arguments using information from the typespec of the function. The main function of this module is the decode function. It is generalized in terms of functions that perform loading and copying of data in order to reuse this function not only for argument decoding in calls, but also for argument decoding in the constructor where arguments are stored in memory, not in calldata. Here is an example of the base case of the decode function:

```
1  def decode(
2    _arg_name,
3    %Type{encoded_type: encoded_type} = type,
4    data_load_fn,
5    _data_copy_fn,
6    calldata_var,
7    _init_calldata_var
8  )
9  when encoded_type > 69 and encoded_type < 102 do
10   """
11   mstore8(memory_offset$, #{type.encoded_type})
12   memory_offset$ := add(memory_offset$, 1)
13   mstore(memory_offset$, #{data_load_fn}(#{calldata_var}))
14   memory_offset$ := add(memory_offset$, #{type.size})
15   #{calldata_var} := add(#{calldata_var}, 32)
16   """
17 end
```

Listing 4.17: Calldata decoding base case

This function clause generates code that copies the value from calldata to memory and store the type for the Bytes<N> ABI type.

Here is an example of the recursive case of the decode function:

```
1  def decode(
```

```
      arg_name,
      %Type{
        encoded_type: 3,
        components: components,
        size: size
      } = type,
      data_load_fn,
      data_copy_fn,
      calldata_var,
      init_calldata_var
    ) do
  tail_offset_var_name = "#{calldata_var}$#{arg_name}"
  init_tail_offset_var_name = "#{init_calldata_var}$#{arg_name}_init"

  """
  mstore8(memory_offset$, #{type.encoded_type})
  memory_offset$ := add(memory_offset$, 1)
  mstore(memory_offset$, #{Enum.count(components)})
  memory_offset$ := add(memory_offset$, 32)

  #{if size == :dynamic do
    """
    let #{tail_offset_var_name} := add(#{init_calldata_var}, #{
      data_load_fn}(#{calldata_var}))
    """
  else
    """
    let #{tail_offset_var_name} := #{calldata_var}
    """
  end}

  let #{init_tail_offset_var_name} := #{tail_offset_var_name}

  #{for {component, index} <- Enum.with_index(components) do
    decode(arg_name <> "#{index}", component, data_load_fn, data_copy_fn
      , tail_offset_var_name, init_tail_offset_var_name)
```

```
36      end}
37      """
38    end
```

Listing 4.18: Calldata decoding recursive case

This function clause recursively generates code that copies the value from calldata to memory and stores type for the tuple ABI type. However, it only considers the tuple structure, i.e., the number of elements in it, and not the types of the elements, each element is decoded recursively.

5. Emitting event

The Event module is responsible for preparing and emitting events. Function emit, which is the main function of the module, takes the event as argument in the following form:

```
1   @type t :: %Event{
2           name: atom(),
3           indexed_arguments: [keyword(Type.t())],
4           data_arguments: [keyword(Type.t())],
5           keccak256: String.t()
6         }
```

Listing 4.19: Event structure

An event has its signature, similar to a method ID for functions, but an event signature is the full Keccak256 hash of the following format:

$$\text{event\_name}(\text{1st\_arg\_type\_name}, \text{2nd\_arg\_type\_name}, ...)$$

Events have indexed and data arguments. Indexed arguments are placed in the topics of the log. Topics are used to filter logs when fetching them from the

blockchain. However, a log can have only up to four topics, with the first one being the event signature, so an event can have up to three indexed arguments. The size of the topics is limited to 32 bytes. For values larger than 32 bytes, its Keccak256 hash is used. The other data could be stored as data arguments that are placed in the data field of the log, with any value stored as is. The module generates code for encoding Elixireum values into indexed and data arguments. The function encode_indexed_argument is responsible for encoding indexed arguments. Here is the clause for types that are smaller than 32 bytes and are placed in topics as is:

```
1  defp encode_indexed_argument(
2    arg_name,
3    %Type{encoded_type: encoded_type} = type,
4    %YulNode{yul_snippet_usage: yul_snippet_usage},
5    compiler_state
6  )
7  when encoded_type not in [1, 3, 102, 103] do
8    var_name = "indexed_#{arg_name}_keccak_var$"
9    arg_name_pointer = "indexed_#{arg_name}_pointer$"
10
11   {"""
12   let #{arg_name_pointer} := #{yul_snippet_usage}
13
14   #{Utils.generate_type_check(arg_name_pointer, encoded_type, "Wrong
       type for indexed argument #{arg_name}", compiler_state.
       uniqueness_provider)}
15
16   #{arg_name_pointer} := add(#{arg_name_pointer}, 1)
17   let #{var_name} := shr(#{8 * (32 - type.size)}, mload(#{
       arg_name_pointer}))
18   """, var_name}
19 end
```

Listing 4.20: Encode word-size indexed argument

Here is the clause for types that do not fit in one word:

```
defp encode_indexed_argument(
       arg_name,
       %Type{} = type,
       %YulNode{yul_snippet_usage: yul_snippet_usage},
       _compiler_state
     ) do
  init_var_name = "indexed_#{arg_name}_keccak_init$"
  var_name = "indexed_#{arg_name}_keccak_var$"
  arg_name_pointer = "indexed_#{arg_name}_pointer$"

  {"""
   let #{init_var_name} := offset$
   let #{arg_name_pointer} := #{yul_snippet_usage}
   #{do_encode_indexed_argument(type, arg_name_pointer, init_var_name, 0)}
   let #{var_name} := keccak256(#{init_var_name}, sub(offset$, #{init_var_name}))
   """, var_name}
end
```

Listing 4.21: Encode compex type indexed argument

The function do_encode_indexed_argument is used to convert and copy Elixireum values to the format for Keccak256 and subsequent use in topics. It is defined recursively, similar to the calldata decode function.

The function encode_data_argument generates code that encodes data arguments, it uses the Return data encoding module 4.3.6.

```
defp encode_data_argument(
       arg_name,
       %Type{} = type,
       %YulNode{yul_snippet_usage: yul_snippet_usage},
       index
     ) do
  """
```

```
8       let #{arg_name}_$ := processed_return_value$
9       let #{arg_name}_init$ := #{arg_name}_$
10      let #{arg_name}_where_to_store_head$ := add(
            processed_return_value_init$, #{index * 32})
11      return_value$ := #{yul_snippet_usage}
12      #{Return.encode(type, "i$", "size$", "#{arg_name}_where_to_store_head$
            ", "where_to_store_head_init$")}
13      """
14    end
```

Listing 4.22: Encode data argument

6. *Return data encoding*

The Return data encoding module is responsible for encoding the value returned by an Elixireum function to an ABI format according to the typespec. It uses a similar approach to the calldata decoding module, but in the opposite way. The main function of this module is encode. Here is the base case of this function for simple types:

```
1   def encode(
2     %Type{encoded_type: encoded_type, size: byte_size},
3     _i_var_name,
4     _size_var_name,
5     where_to_store_head_var_name,
6     _where_to_store_head_init_var_name
7   )
8   when encoded_type > 69 and encoded_type < 102 do
9     offset = 8 * (32 - byte_size)
10
11    """
12    #{Utils.generate_type_check(...)}
13
14    return_value$ := add(return_value$, 1)
15
```

```
    mstore(#{where_to_store_head_var_name}, shl(#{offset}, shr(#{offset},
        mload(return_value$))))
    #{where_to_store_head_var_name} := add(#{where_to_store_head_var_name}
        , 32)

    return_value$ := add(return_value$, #{byte_size})
    """
  end
```

Listing 4.23: Encode base case

Here is an example of a recursive case:

```
  def encode(
        %Type{
          encoded_type: 103 = encoded_type,
          items_count: size,
          components: [components]
        },
        i_var_name,
        size_var_name,
        where_to_store_head_var_name,
        where_to_store_head_init_var_name
      )
      when is_integer(size) do
    """
    #{Utils.generate_type_check(...)}

    return_value$ := add(return_value$, 1)
    let #{size_var_name} := mload(return_value$)

    #{Utils.generate_value_check(...)}

    return_value$ := add(return_value$, 32)

    for { let #{i_var_name} := 0 } lt(#{i_var_name}, #{size_var_name}) { #
        {i_var_name} := add(#{i_var_name}, 1) } {
```

```
24        #{encode(components, i_var_name <> "_", size_var_name <> "_",
              where_to_store_head_var_name, where_to_store_head_init_var_name)
              }
25      }
26      """
27    end
```

Listing 4.24: Encode recursive case

Chapter 5

Results and Discussion

This chapter presents an overview of results. Section 5.1 provides results we got during assessment of the Elixireum. Section 5.2 contains an interpretation of results.

5.1 Results and Comparison

We conducted a series of tests by applying the test suite to a set of prepared smart contracts. All tests concluded successfully, signifying the achievement of the main project goal: Elixireum smart contracts successfully compile and deploy to the Ethereum blockchain. We then proceeded to compare Elixireum with Solidity. We benchmarked all the public functions of the ERC-20 implementation in Elixireum and its full analogue in Solidity, and conducted the same benchmarking for the ERC-721 implementation. Tables IV and V present the detailed metrics that we obtained.

The comparison reveals that while Elixireum successfully implements the functionalities of ERC-20 and ERC-721 contracts, it generally consumes more

Method	Elixireum Gas Used	Solidity Gas Used
mint	58197.4	54799.6
approve	47619.8	46570.8
transferFrom	54812.0	48492.2
transfer	55659.8	51813.4
burn	33061.6	29433.25
deploy of contract	1939441	732543

TABLE IV
Gas usage comparison between Elixireum and Solidity for ERC-20 contract (20 runs)

Method	Elixireum Gas Used	Solidity Gas Used
setApprovalForAll	46596.0	46304.0
mint	166817.4	162675.4
transferFrom	87123.0	55652.6
approve	54798.0	49007.8
burn	58539.6	29944.0
deploy of contract	3067454	1113786

TABLE V
Gas usage comparison between Elixireum and Solidity for ERC-721 contract (20 runs)

gas compared to Solidity for the same operations. Table IV shows that, on average, Elixireum is 8% less efficient than Solidity for function calls of ERC-20 methods and approximately 164% less efficient in terms of gas consumption during the deployment process. Table V indicates even less favorable metrics: Elixireum is about 20.5% less efficient for ERC-721 methods and consumes 175% more gas during deployment. These findings indicate that while Elixireum is functional and compatible with the Ethereum blockchain, there is a need for optimization to reduce gas consumption and improve efficiency.

5.2 Discussion

The findings from this chapter contribute significantly to the broader thesis goal of assessing the feasibility and potential of Elixireum in the context of smart contract development. We confirmed our hypothesis that an Elixir-like language can be compiled to EVM.

Moreover, we conducted a comprehensive comparison of gas consumption between Elixireum and Solidity. While the results showed that Elixireum consumes 2.7 times more gas than Solidity for deployment, the gas usage for ERC-20 and ERC-721 methods is comparable. This demonstrates that it is possible to have a functional language with dynamic typing and types stored in memory that consumes approximately 14% more gas than Solidity.

With the rise of rollup chains mentioned in Chapter 1, the tradeoff between gas consumption and development ease should be considered. While a call of contract written with Elixireum may cost $1.15 instead of $1 with Solidity due to higher gas consumption, the ease and speed of development with Elixireum can lead to earlier deployment. This earlier deployment can attract more users, potentially outweighing the slight increase in gas costs.

Chapter 6

Conclusion

We designed and implemented the Elixireum language along with its compiler. The Elixireum compiler and language are capable of supporting ERC-20 and ERC-721 standards, and we conducted measurements to compare our language with Solidity.

6.1 Limitations

The scope of this study was limited to the functionality that we considered most important to demonstrate the feasibility of Elixireum. Consequently, several features were left out of scope:

1. **External libraries:** External libraries are not implemented. This limitation prevents the reuse of the same functionality across multiple smart contracts and complicates the development of complex smart contracts, requiring redeployment of the same functionality and consequently requires more gas.

2. **External smart contract interaction:** Interaction with external smart contracts is not implemented. This limitation means that Elixireum cannot currently communicate with other contracts, making it impossible to implement popular patterns such as proxies or factories.

3. **Multiple files or modules:** Elixireum development is limited to a single file, making it impossible to reuse any Elixireum functionality from other smart contracts or to integrate additional functionalities from external sources.

4. **High deployment gas consumption:** The deployment of Elixireum contracts results in high gas consumption.

5. **Macros support:** We decided to exclude macros support from our initial scope due to the complexity of implementing this feature and time constraints.

6. **Pattern matching:** This feature was also excluded from the scope due to the similar reasons.

6.2 Future Work

We propose the following directions for future work and development of Elixireum:

1. **Runtime mapping:** Implementing mapping runtime on top of the current storage module using EVM transient storage. Currently, Solidity does not support mappings in memory, only in storage, so this feature could be highly innovative and novel.

2. **Reducing deployment gas consumption:** The high gas consumption of deployment is likely connected to the large amount of generated Yul code. We believe it is possible to reduce these costs by factoring out similar Yul code snippets into standard functions.

Bibliography cited

[1] D. Harz and W. J. Knottenbelt, "Towards safer smart contracts: A survey of languages and verification methods," *ArXiv*, vol. abs/1809.09805, 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:52844161.

[2] V. Buterin. "Ethereum whitepaper." (2014), [Online]. Available: https://ethereum.org/en/whitepaper.

[3] "Smart contract languages." Accessed Oct. 22, 2023. (2022), [Online]. Available: https://ethereum.org/en/developers/docs/smart-contracts/languages/.

[4] W. Zou, D. Lo, P. S. Kochhar, *et al.*, "Smart contract development: Challenges and opportunities," *IEEE Transactions on Software Engineering*, vol. 47, no. 10, pp. 2084–2106, 2021. DOI: 10.1109/TSE.2019.2942301.

[5] "Language influences." Accessed Oct. 29, 2023. (2021), [Online]. Available: https://docs.soliditylang.org/en/v0.8.22/language-influences.html.

[6] "Solidity." Accessed Oct. 29, 2023. (2023), [Online]. Available: https://docs.soliditylang.org/en/v0.8.22/.

[7] "Vyper." Accessed Oct. 29, 2023. (2020), [Online]. Available: `https://docs.vyperlang.org/en/stable/`.

[8] "Elixir." (2023), [Online]. Available: `https://elixir-lang.org/` (visited on 11/14/2023).

[9] S. O'Grady. "The redmonk programming language rankings: January 2023." Accessed Oct. 29, 2023. (2023), [Online]. Available: `https://redmonk.com/sogrady/2023/05/16/language-rankings-1-23/`.

[10] G. G. William M. Waite, *Compiler Construction*. 1984.