Theia v0.1a white paper

Authors

George S.

Kirill Z.

March 2023

Table of Contents

Introduction

With the rapid development of decentralized applications and the emergence of the Web 3.0 era, there is a growing demand for programming languages that can facilitate the creation of efficient and reliable blockchain-based systems. Theia is a language specifically designed to create applications for the Unit blockchain, which aims to merge all the existing spheres of Web 3.0, including decentralized finance, gaming, and social media.

Theia is a highly versatile programming language that supports UTF-8, enabling developers to create applications that can handle complex multi-lingual data. Theia's flexibility is attributed to its virtual machine, which allows it to run on various platforms and in a variety of use cases. Additionally, Theia can be compiled directly to assembly, making it highly efficient and fast.

Inspired by some of the most widely used programming languages, including Java, Python, and C++, Theia provides a simple and user-friendly programming environment that is easy to learn and use. Furthermore, Theia supports Solidity code injections, which makes it highly compatible with other blockchain technologies.

In this paper, we explore the capabilities and features of Theia as a programming language for developing blockchain-based applications. We discuss the syntax and semantics of Theia, its performance, and its suitability for various use cases. We also highlight the key advantages of using Theia in blockchain development, including its simplicity, flexibility, and efficiency.

1. Our goals

In the realm of blockchain technology, various programming languages have been modified or developed to cater to the specific needs of this niche. For instance, Solidity and C++ are popular languages used in blockchain development, which can handle the basic requirements of the blockchain ecosystem. However, the Unit blockchain offers a more complex solution with its Virtual File System, Virtual Data Storage, Virtual Machine, and truly WEB3.0 philosophy.

In Unit, the programming environment must satisfy all aspects of the product to achieve efficiency, user-friendliness, and a positive development experience. As such, the programming language utilized in Unit blockchain development must be designed to cater to the unique requirements of this system.

While Solidity and C++ are great examples of programming languages that have been successful in the blockchain sphere, the Unit blockchain requires a programming language that can handle its unique features and optimize its development process.

2. Syntax

Theia's syntax is similar to popular programming languages such as Java and C++. It is designed to be flexible and versatile, supporting a wide range of coding styles and programming paradigms, including object-oriented, imperative, and functional programming. The syntax of Theia is carefully crafted to balance simplicity, readability, and expressiveness.

2.1 Classes

Theia is its support for classes, a fundamental building block of object-oriented programming. While classes are a common feature in traditional programming languages, they are a relatively new concept in blockchain languages.

In Theia, classes are similar to contracts in Solidity, the language used to develop smart contracts on the Ethereum blockchain. However, Theia's class system has more features than Solidity's contract system.

Classes in Theia provide a way to define user-defined data types that encapsulate data and behavior. They allow developers to organize their code into logical Units, making it easier to manage and maintain. Theia's class system supports inheritance, which allows developers to create classes that inherit properties and methods from other classes. This helps to reduce code duplication and improve code reuse.

In addition to inheritance, Theia's class system also supports interfaces, which define a set of properties and methods that a class must implement. This helps to ensure that classes adhere to a common interface, making it easier to integrate different components of an application.

Another important feature of Theia's class system is its support for access modifiers, such as private, public, and protected. These modifiers control the visibility of properties and methods within a class, providing a way to encapsulate data and behavior.

Example of Theia's class:

```
/* Example of class */
class A {
    public:
    ...
    private:
    ...
    protected:
    ...
}
```

Main difference between Solidity's contracts and Theia classes is initialization. The data that comprises the class is per-instantiation. In other words `classInstance1` and `classInstance2` each have their own state. In Solidity there is only one contract in the program. In this sense the state of the contract is a singleton. All the state is program-wide level. It's as if the entire class were global. Another way of saying this is that all the data fields of the contract are like C++ static members. But if class is not a static member there isn't ability to share states between blockchain nodes. To resolve it you can add a single keyword `contract` to the class and it will work as traditional Solidity contract.

Example of contract-class in Theia:

```
/* Example of contract-class */
contract class B {
    public:
    ...
    private:
    ...
    protected:
    ...
}
```

This approach will reduce vulnerabilities in DAO and will help with efficient development.

2.2 Functions

In Theia, functions serve a similar purpose as they do in C++, Solidity or Python. They are used to encapsulate a set of instructions and execute them whenever needed. In contract-classes functions that modified values must be marked as `paid` because they mutate contract states and consume gas.

There are several ways to declare functions in Theia:

1. C-style declaration. This is the most common way to declare a function in Theia. It involves specifying the function name, parameters, and return type (if any). For example:

   ```
   /* typical function declaration */
   bool func1(int a, int b)
   {
       return a > b;
   }
   ```

2. But we can reduce keywords in this declaration. Theia is type-safe and return statement can be known without type keyword needing:

   ```
   /* function declaration without type keyword */
   func2(int a, int b)
   {
       return a > b;
   }
   ```

   Boolean operator `>` means that returns type will be `bool` and pre-compiler will add this type to the source code. Be aware, this function does NOT have dynamic type!

3. Function Expression. This involves assigning a function to a variable, similar to how values are assigned to variables. For example:

   ```
   /* boolean function expression */
   const function func3 = (int a, int b) { return a > b };
   ```

   This creates a function called `func3` and assigns it to a constant variable. The function can then be called using the variable name, like this:

```
/* function call */
bool res = func3(5, 10); /* res will be false */
```

4. Anonymous (Lambda) Functions. Anonymous functions are a powerful tool in programming, as they allow for the creation of small, one-time use functions that do not require a name or a separate file to be saved in. They are commonly used in functional programming, where functions are treated as first-class citizens and can be passed as arguments to other functions or used as return values. Example:

```
/* common function */
/* we will apply filter to the result */
/* filter-function will be passed as lambda */
int calculator(int a, int b, function filter)
{
    /* call filter and return its result */
    return filter(a * b);
}

int main()
{
    /* call calculator function with filter */
    /*                              ------------lambda function*/
    /*                              ---------------------------*/
    int result = calculator(-5, 10, (int v)=>{ return v + 10; } );
}
```

2.3 Variables

In blockchain programming, Theia offers a wide range of variable types that are commonly used in programming languages. These variables can be declared as constants or can be classified as public, private, or protected.

Constant variables are those whose values remain unchanged throughout the program's execution. They can be useful in situations where a fixed value needs to be used repeatedly in the code, such as when defining mathematical constants or setting predefined limits.

Theia also allows for the use of public, private, and protected variables. Public variables can be accessed and modified from any part of the code, while private variables are only accessible within the same class or module. Protected variables, on the other hand, can be accessed within the same class or module, as well as by any subclasses.

Variables in Theia have access to the upper and same scope where they are declared. This means that they can be accessed and modified within the same function or block of code where they were defined, as well as in any nested functions or blocks of code.

Properly declaring and using variables is essential for efficient and effective programming in blockchain. Theia's support for common variable types and access levels allows developers to create more secure and structured code, ultimately leading to more reliable and robust blockchain applications.

2.4 Literals

Literals are values that are directly assigned to a variable or used in an expression, such as numbers or strings, as opposed to being the result of a variable or expression evaluation. To increase computational efficiency, the Theia enables the grouping of integer literals as a single integer.

2.5 Conditionals

The Theia programming language offers two options for implementing conditionals, which are the ternary operator (*condition ? true : false*) and the conventional *if* statement.

2.6 Loops

Theia has two loop types, namely *for* and *while*, which can be used to execute a block of code repeatedly. They can be optimized in the compiler to enhance their efficiency during execution.

2.7 Data types

All native data types in Theia:

Type	Note
bool	**true/flase**
byte	signed type from **-128** to **127**, inclusive
short	signed type from **-32768** to **32767**, inclusive
int	signed type from **-2147483648** to **2147483647**, inclusive
long	signed type from **-9223372036854775808** to **9223372036854775807**, inclusive
uint32	unsigned type from **0** to **4294967295**
uint64	unsigned type from **0** to **18446744073709551615**
uint128	unsigned type from **0** to **340282366920938463463374607431768211455**
unit256	unsigned type from **0** to **115792089237316195423570985008687907852589419931798687112530834793049593217025**
double	in range: **1.7E+/-308**
char	'**\u0000**' to '**\uffff**' inclusive or from **0** to **65535**

All non-native data types in Theia:

Type	Note
waddress	represents a 26-byte value. It is used to store the Unit address of an account or contract on the blockchain
string	contains a collection of characters in UTF-8 encoding
array	store multiple values in a single variable
object	root of the class hierarchy. Every class has Object as a superclass.

2.8 Waddress

In Theia, an address is a variable type that represents a 26-byte value, typically written in hexadecimal format. It is used to store the Unit address of an account or contract on the blockchain.

Addresses are used extensively in Theia code to interact with other contracts or to send and receive UNT (the native cryptocurrency of the Unit network). For example, if you want to send UNT to another address, you would use the `transfer()` function, passing in the recipient's address and the amount of UNT you want to send.

Exapmle:

```
/* creates new waddress */
waddress wallet = “UNTxFgkveYokZXbFG4XwXigtzsh4JEP”;

/* transfer 10 tokens */
transfer(wallet, 10);
```

2.9 Strings

In Theia, a string is a sequence of characters. It is a data type that is used to store text.

The string implementation in Theia exhibits comparable potency to that of Java or Python. This attribute is attributed to their memory-efficient nature and the availability of compiler optimization techniques.

Example:

```
/* creates new string */
string text = “Hello, World!”;
```

2.10 Arrays

Theia's arrays are similar to Java arrays in terms of syntax and functionality. In addition to providing a familiar programming experience for Java developers, Theia's arrays also offer the added benefit of working as hash-tables.

Hash-tables are a data structure that allows for efficient data storage and retrieval by using a key-value pairing system. With Theia's array implementation, developers can use the index of an element as the key and the value stored at that index as the corresponding value.

This means that developers can easily store and retrieve data in Theia's arrays using a simple indexing system, similar to how they would access values in a Java array. Additionally, the hash-table functionality provides a fast and efficient way to perform lookups and other operations on the array's data.

Example:

```
/* fixed size array of integers */
int array1[10];
array1[5] = 99; /* fifth element will be 99 now */

/* dynamic size array of integers */
int array2[];
array2[5] = 66; /* fifth element will be created and assigned with value 66 */
/* elements with indexes from 0 to 4 will be still null */
```

2.11 Object

Theia provides the same meaning as the Object class in Java. In Java, the Object class serves as the parent class of all other classes and is the most fundamental class in the Java class hierarchy. It defines the basic properties and behaviors that all objects in Java inherit.

Similarly, in Theia, the Object class is also the base class for all other classes and serves as a blueprint for defining objects. It provides the basic features and characteristics that all objects in Theia possess, such as the ability to compare objects and retrieve their hash code.

Overall, the similarity between the Object class in Java and Theia is crucial as it ensures consistency and familiarity between the two programming languages, making it easier for programmers to transition between the two.

3. Design of Theia Compiler

The Theia Compiler, abbreviated as TC, is a crucial component of the Theia programming environment. Its primary goal is to optimize the memory and CPU performance of code in order to reduce the usage of gas. Gas is the transaction fee required to execute smart contracts on the blockchain, and reducing its usage can lead to significant cost savings.

The TC achieves this by analyzing the code and implementing various optimizations that improve its memory and CPU performance. This results in code that requires less gas to execute, making it more efficient and cost-effective.

The TC also provides a proof of generation. This is useful in scenarios where the source code is required to be verified, such as in auditing and compliance checks. It is also necessary for blockchain smart-contract system.

3.1 Lexing

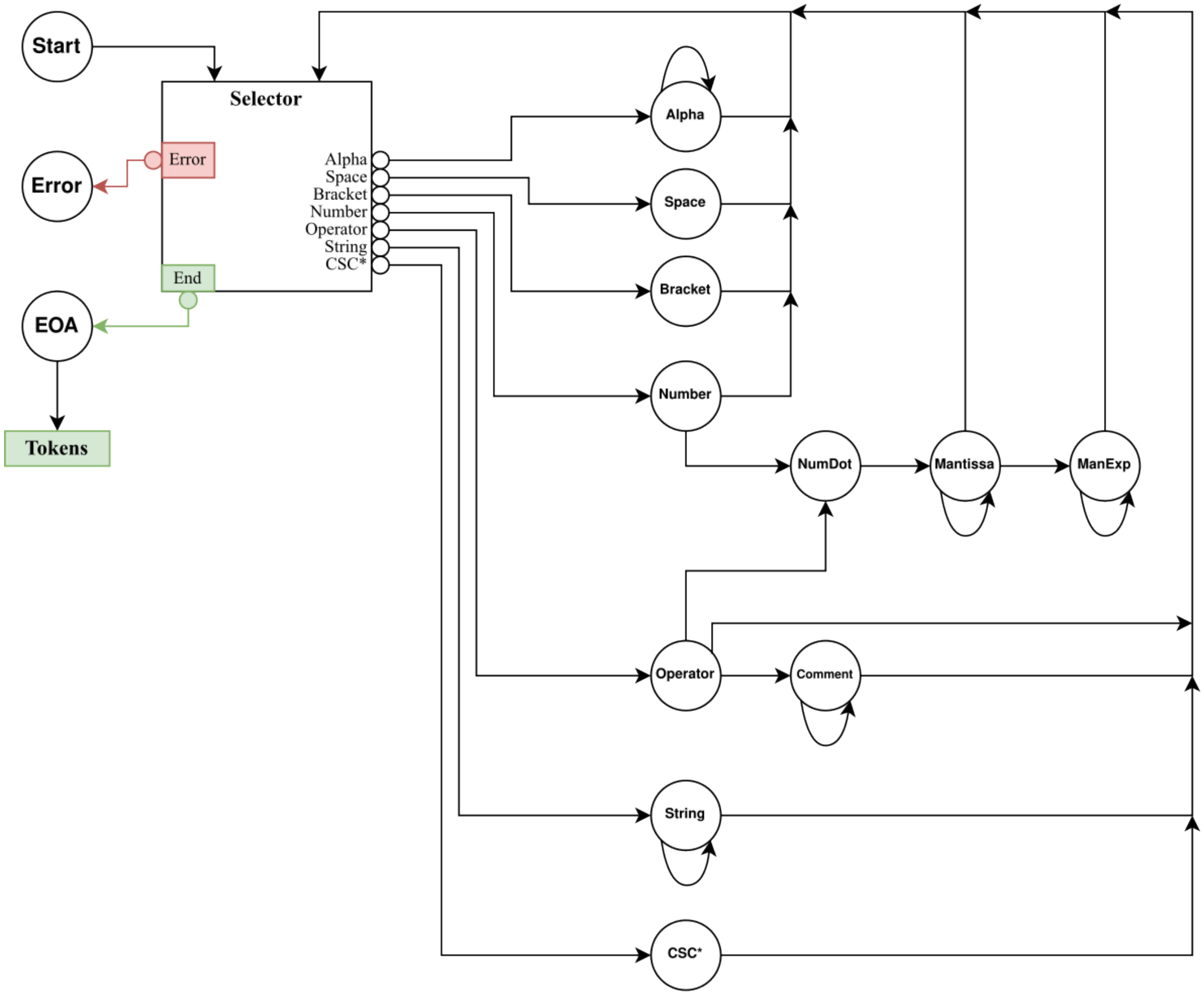
As a programming language, Theia must undergo a compilation process to convert the source code into machine-readable instructions that can be executed by a computer. One of the most critical steps in this process is tokenization, where the raw text of the code is broken down into individual tokens. However, this can be challenging as the presence of raw text in the code can lead to errors and inaccuracies.

To address this issue, Theia uses a lexer based on a Finite State Machine (FSM). This lexer is designed to handle all tokens and can detect token errors, such as mistakes in keywords, to ensure that the code is correctly tokenized.

An FSM-based lexer works by defining a set of states and transitions that represent the grammar of the programming language. As the source code is processed, the lexer transitions between these states to identify the corresponding tokens accurately. For example, in Theia, the lexer would define states to represent keywords such as "if", "else", and "while," allowing for efficient and accurate tokenization of the code.

One of the significant advantages of using an FSM-based lexer in Theia is its ability to detect token errors accurately. By using a finite set of states to represent the different parts of the grammar, the lexer can ensure that each token is correctly identified and that errors, such as mistakes in keywords, are detected and resolved.

In addition to improving the accuracy of tokenization, an FSM-based lexer can also enhance the performance of the compiler. This is because FSM-based lexers are typically faster and more efficient than other types of lexers, such as regular expression-based lexers.



***CSC - Comma, Semicolon, Colon**

Figure 1. Lexer's finite state machine scheme.

3.2 Parsing

One of the significant challenges faced by compiler developers is handling complex grammars. A grammar is considered complex if it has ambiguous or non-context-free constructs, making it difficult to parse using traditional parsing techniques such as LL or LR parsing. To address this challenge, the TC uses the GLR (Generalized LR) algorithm, a powerful parsing algorithm that can handle complex grammars.

Main advantage of the GLR parsing algorithm is its ability to handle ambiguous grammars without requiring the grammar to be simplified or transformed. This makes the GLR algorithm a popular choice for programming languages with complex or evolving grammars, such as C++ and Python. Additionally, the parse forest produced by the GLR algorithm can be used for advanced features, such as syntax highlighting, error reporting, and refactoring.

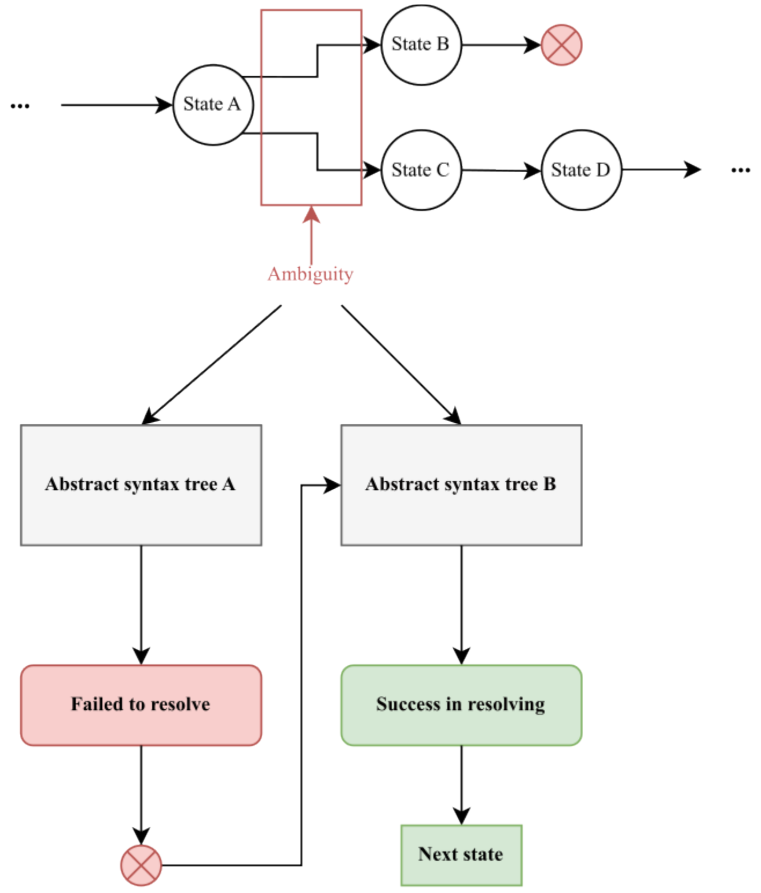


Figure 2. Simple example of GLR parser.

3.3 Proof of generation

In order to ensure the authenticity and integrity of generated contracts, it is imperative that each compiled contract undergoes a process of hashing. Hashing, which involves generating a unique fixed-length representation of the contract data, serves as a means of verifying that the contract has been generated in a secure and accurate manner. As such, the hashing of compiled contracts represents a critical component of the overall quality control and assurance measures that are implemented within contract generation processes.

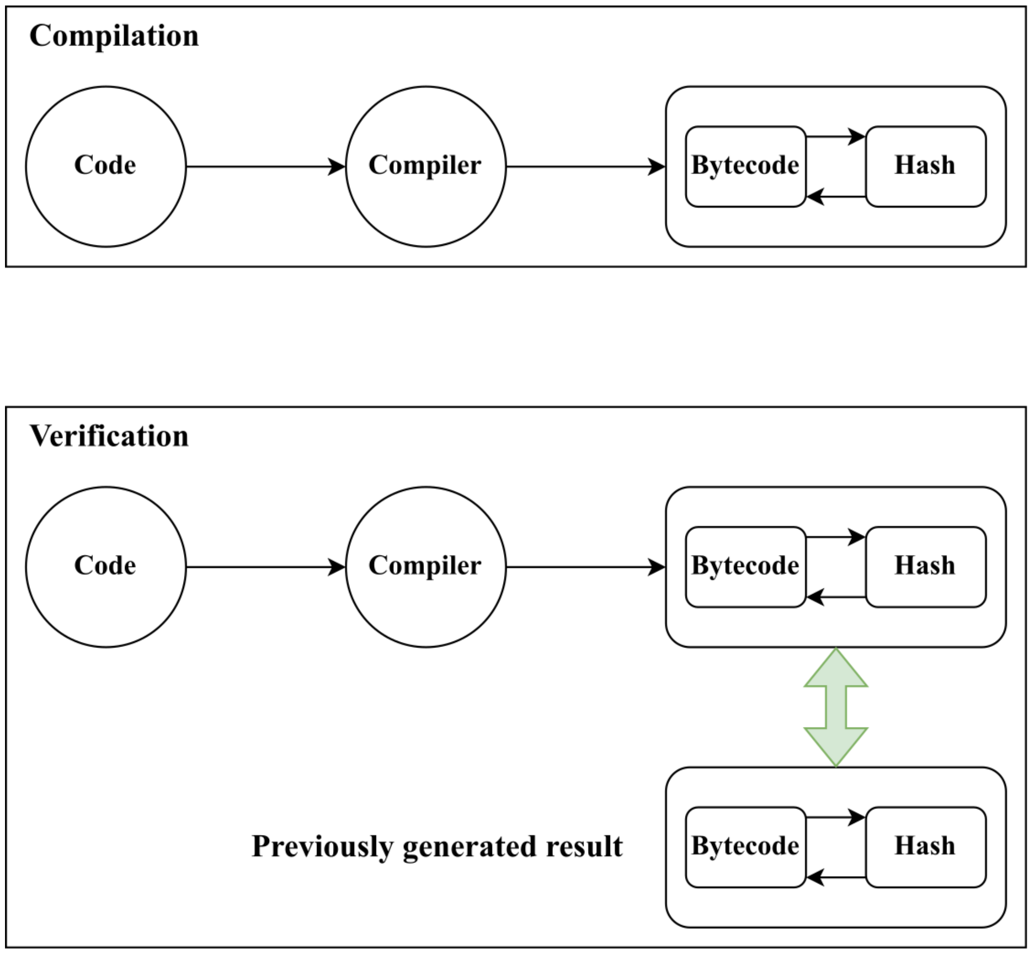


Figure 3. Compilation and code verification.

4. Future work

Our work on Theia is just beginning, and there are a number of exciting directions to improve both the Theia language and compiler. We look to expand Theia in the following directions:

Implementation of GLR optimization: in worst case GLR algorithm gives O(n^3) complexity, so we need to work on decreasing such states.

Syntax improvement: add more syntax features to give the most friendly experience. For example, create LINQ-like syntax to simplify data management.

Implement CUDA and Tensor Flow support: we truly think, that blockchain is not only about finance!

Solidity code injections: it enables developers to seamlessly integrate Solidity code injections into their programming projects, thereby expanding the range of available functionalities.

5. References