Lolisa: Formal Syntax and Semantics for a Subset of the Solidity Programming Language

Zheng Yang\textsuperscript{1}\textsuperscript{*} \hspace{1cm} Hang Lei
zyang.uestc@gmail.com \hspace{1cm} hlei@uestc.edu.cn

\textsuperscript{1}School of Information and Software Engineering, University of Electronic Science and Technology of China, No.4, Section 2, North Jianshe Road, 610054, Sichuan, Chengdu, P.R. China.

Abstract This article presents the formal syntax and semantics for a large subset of the Solidity programming language developed for the Ethereum blockchain platform. This subset is denoted as Lolisa, which, to our knowledge, is the first mechanized and validated formal syntax and semantics developed for Solidity. The formal syntax of Lolisa adopts a stronger static type system than Solidity for enhanced type safety. In addition, Lolisa not only includes nearly all the syntax components of Solidity, such as mapping, modifier, contract, and address types, but it also contains general-purpose programming language features, such as multiple return values, pointer arithmetic, struct, and field access. Therefore, the inherent compatibility of Lolisa allows Solidity programs to be directly translated into Lolisa with a line-by-line correspondence without rebuilding or abstracting, and, in addition, the inherent generality of Lolisa allows it to be extended to express other programming languages as well. To this end, we also present a preliminary scheme for extending Lolisa to other languages systematically. In recent work, we developed a general, extensible, and reusable formal memory (GERM) framework that can simultaneously support different formal verification specifications, particularly at the code level, for verifying the properties of programs based on higher-order logic theorem proving. The framework simulates physical memory hardware structure, including a low-level formal memory space, and provides a set of simple, nonintrusive application programming interfaces and assistant tools using Coq that can support different formal verification specifications simultaneously. The proposed GERM framework is independent and customizable, and was verified entirely in Coq. We also developed an extension of Curry-Howard isomorphism, denoted as execution-verification isomorphism (EVI), which combines symbolic execution and theorem proving for increasing the degree of automation in higher-order logic theorem-proving assistant tools. To capitalize on these breakthroughs, the semantics of Lolisa follows EVI, and is designed based on the GERM framework using natural semantics that observes both terminating and diverging executions. Therefore, in conjunction with the formal interpreter developed based on Coq in the present work, it is then theoretically possible for programs written in Lolisa to be symbolically executed in higher-order theorem-proving assistants directly, and have their properties verified automatically at the same time. The semantics of Lolisa are validated, and we certify that Lolisa satisfies EVI. This work is part of our project to build a general and powerful formal symbolic process virtual machine for certifying and verifying smart contracts operating on the blockchain platform easily and automatically without consistency problems.

Keyword: formal verification; programming language; Solidity; Coq; Ethereum; Blockchain

1. Introduction

The blockchain platform [1] is one of the emerging technologies developed to address a wide range of disparate problems, such as those associated with cryptocurrency [6] and distributed storage [14]. Ethereum is one of the most widely adopted blockchain systems. One of the most important features of Ethereum is that it implements a general-purpose Turing-complete programming language denoted as Solidity [7]. This allows for the development of arbitrary applications and scripts that can be executed in a virtual runtime environment denoted as the Ethereum Virtual Machine (EVM) to conduct blockchain transactions automatically. These applications and scripts (i.e., programs) are collectively denoted as smart contracts. The growing use of smart contracts has led to increased scrutiny of their security. Smart contracts can include particular properties (i.e., bugs) making them susceptible to deliberate attacks that can result in direct economic loss. Some of the largest attacks on smart contracts are well known, such as the attack on decentralized autonomous organization (DAO) [9] and Parity wallet [10] contracts. In fact, many classes of subtle bugs, ranging from transaction-ordering dependencies to mishandled exceptions, exist in smart contracts [11].

One of the challenges that must be confronted in the development of smart contracts is that the programming process differs from that of normal programs. Smart contracts are a special type of digital contract, which means the source code is the law, so, similar to the contracts written in natural language, the obligations and terms should be presented in the smart contracts explicitly. However, smart contracts developers are generally programmers, whose grasp of obligations and terms is secondary to their grasp of programming, rather than legal experts. So, even
though programming language is a type of formal language, there are many loopholes of law in smart contracts caused by the programming habits of programmers. An additional challenge is that conventional software testing methods, such as α and β testing, do not perform well for smart contracts because of the near impossibility of patching a contract once it has been deployed due to the anonymous Byzantine execution environment of a public blockchain. Besides, these loopholes of contracts are different from common bugs to be easily found out. Because these loopholes often directly cause the economic loss stealthily instead of making the programs break off. Moreover, software engineering techniques employing such static and dynamic analysis tools as Manticore [12] and Mythril [13] have not yet been proven to be effective at increasing the reliability of smart contracts.

Due to these difficulties, formal verification for the blockchain platform has become a subject of particular interest in recent years because it is one the most powerful program verification technologies. And many popular research works focus on the formal verification on the bytecode of Solidity. For example, KEVM [18] is a formal semantics for the EVM written using the K-framework, like the formalization conducted in Lem [19]. KEVM is executable, and therefore can run the validation test suite provided by the Ethereum foundation. The symbolic reasoning conducted for KEVM programs involves specifying properties in Reachability Logic, and verifying them with a separate analysis tool. While these represent currently available mechanized formalizations of operational semantics, axiomatic semantics, and formal low-level programming verification tools for EVM and Solidity bytecode, they are not well suited to high-level programming languages, such as Solidity. In response, the Ethereum community has placed open calls for formal verification proposals [15] as part of a concerted effort to develop formal verification strategies [16].

Actually, in other field of computer science, a number of interesting studies have focused on developing mechanized formalizations of operational semantics for different high-level programming languages. For the C language, the Cholera project [32] represents one of the most interesting works that formalized the static and dynamic semantics of a large subset of C using the HOL proof assistant. The CompCert project [33] is another influential verification work for C and GCC that developed a formal semantics for a subset of C denoted as Clight. This work formed the basis for VST [34] and CompCertX [35]. In addition, a number of interesting formal verification studies have been conducted for operating systems based on the CompCert project. Tews et al. [36] developed a denotational semantics for a subset of the C++ language, which were presented as a shallow embedding in the PVS prover. For Java, the work of Igarashi et al. [37] is particularly inspiring because it presents a minimal core calculus for Java and Generic Java (GJ), and the important core properties are certified. Another similar work has been conducted for proving Java type soundness [38]. In addition, the operational semantics of JavaScript have been investigated [39], which is of particular importance to the present study because Solidity is a programming language like JavaScript. However, not all these frameworks can be executed in higher-order logic theorem-proving assistants directly.

In addition, the formal syntax and semantics of programming languages play a very important role in several areas of computer science, particularly in program verification. For advanced programmers and compiler developers, formal semantics provide a more precise alternative to the informal English descriptions that usually pass as language standards. In the context of formal methods, such as static analysis, model checking, and program proof, formal semantics are required to validate the abstract interpretations and program logic (e.g., axiomatic semantics) used to analyze and verify programs. The verification of programming tools such as compilers, type-checkers, static analyzers, and program verifiers is another area where formal semantics for the languages involved is a prerequisite. However, the development of high-level formal specifications of Solidity and relevant formal verification tools have attracted considerably less interest from researchers despite its importance for programming and debugging smart contract software. Therefore, the need for related work is very urgent to fill this gap.

Higher-order logic theorem proving is one of the most rigorous technologies for verifying the properties of programs. This involves establishing a formal model of a software system, and then verifying the system according to a mathematical proof of the formal model. However, numerous problems regarding reusability, consistency, and automation are encountered when applying theorem-proving technology to program verification. One of the available solutions for addressing these problems is to design a formal symbolic process virtual machine (FSPVM) in a higher-order theorem-proving system such that real-world programs can be symbolically executed, and their properties verified automatically using the execution result. However, the successful implementation of such an FSPVM must overcome a number of challenges. In a recent work [3], we addressed some of these challenges by developing a general, extensible, and reusable formal memory (GERM) framework based on higher-order logic using Coq. It includes a formal memory space, and provides a set of simple and nonintrusive memory management application programming interfaces (APIs) and a set of assistant tools. The GERM framework can express the interaction relationships between special and normal memory blocks. One the one hand, the framework functions independently of higher level specifications, so it can be used to represent intermediate states of any high-level specifications designated by general users, which facilitates the reuse of intermediate representations in different high-level formal verification models. On the other hand, the framework can be used as an operating environment to facilitate automated higher-order logic theorem proving. We also developed an extension of Curry-Howard isomorphism (CHI), denoted herein as execution-verification isomorphism (EVI), which can combine theorem proving and symbolic execution technology in the operating
environment of the GERM framework to facilitate automated higher-order logic theorem proving. The use of EVI makes it possible to execute a real-world program logically while simultaneously verifying the properties of the program automatically in Coq or using another proof assistant that supports higher-order logic proving based on CHI without suffering consistency problems.

The work presented in this paper also aims to address an additional challenge associated with designing an FSPVM in a higher-order theorem-proving system, as part of our ongoing project to build a general and powerful FSPVM for certifying and verifying smart contracts operating on multiple blockchain platforms. This challenge involves formalizing real-world programming languages as an extensible intermediate programming language (IPL), and integrating the IPL into the logic operating environment. Here, the formal syntax and semantics of the IPL should be equivalent to those of the respective real-world target programming languages. Such an IPL contributes to addressing the reusability and consistency problems because it standardizes the process of building a formal model for programs. As such, the equivalent formal versions of programs written in the IPL can serve as their formal models without additional abstracting or rebuilding. The developed formal interpreter based on the GERM framework and EVI should be able to automatically execute the formal version of programs written in the IPL in the logic operating environment. Therefore, the present article capitalizes upon our past work by defining the formal syntax and operational semantics for a large subset of the Solidity programming language. This subset is denoted herein as Lolisa, which, to our knowledge, is the first mechanized and validated formal syntax and semantics developed for Solidity. Lolisa has the following features.

- **Consistency** Lolisa formalizes most of the types, operators, and mechanisms of Solidity, including reference arithmetic, references to functions, and struct, and address, and mapping types, as well as the contract inheritance mechanism, and it includes nearly all of the Solidity syntax according to Solidity documentation [21]. In addition, we build a standard library based on Lolisa to represent the built-in data structures and functions of EVM, such as msg, block, and send. As such, programs written in Solidity can be translated into Lolisa, and vice versa, with a line-by-line correspondence without rebuilding or abstracting, which are operations that can negatively impact consistency.

- **Static Type System** The formal syntax in Lolisa is defined using generalized algebraic datatypes (GADTs) [2], which imparts static type annotation to all the values and expressions of Lolisa. In this way, Lolisa has a stronger static type system than Solidity for checking the construction of programs. As such, it is impossible to construct ill-typed terms in Lolisa, which also assists in discovering ill-typed terms in Solidity source code. Moreover, the formal syntax ensures that all expressions and values in Lolisa are deterministic.

- **Executable and Provable** In contrast to similar efforts focused on building formal syntax and semantics for high-level programming languages, the formal semantics of Lolisa are defined based on the GERM framework in conjunction with EVI. As such, it is theoretically possible for programs written in Lolisa to be symbolically executed and have their properties simultaneously verified automatically in higher-order logic theorem-proving assistants directly as program execution in the real world when conducted in conjunction with a formal interpreter.

- **Mechanized and Validated** The syntax and semantics of Lolisa are mechanized using the Coq proof assistant [5]. We also develop a formal verified interpreter in Coq to validate whether Lolisa satisfies the above **Executable and Provable** feature and the meta-properties of the semantics. The details regarding the implementation of our formal interpreter will be presented in our next paper.

- **Extensible and Universal** Although Lolisa is designed for Solidity, it includes many general features of other general-purpose high-level programming languages. As such, the core syntax and semantics of Lolisa can be extended to formalize other similar programming languages. We therefore provide a preliminary scheme to systematically extend Lolisa to support different high-level programming languages.

The remainder of this paper is structured as follows. Section 2 introduces preparatory work about the modification of the GERM framework to support the semantics of Lolisa. Section 3 elaborates on the formal abstract syntax of Lolisa, and compares this with the formal abstract syntax of Solidity. Section 4 presents the module system of Lolisa to formalize the behavior of contract inheritance and member access. Section 5 presents the formal dynamic semantics of Lolisa, including the program execution semantics and the formal standard library for the built-in data structures and functions of EVM. Section 6 describes the integration of the Lolisa programming language and its semantics within the formal verified interpreter and its validation. Section 7 introduces the extensibility and universality of Lolisa, and proposes a preliminary scheme to systematically extend Lolisa to support different high-level programming languages. Finally, Section 8 presents the conclusions of our work.

2. Preparation Work

Before defining the formal specifications of Lolisa, it is necessary to achieve the preparation works for defining the basic environment.
2.1 Predifinitions

Tables 1 summarize the helper functions used in the dynamic semantic definitions. State functions calculate commonly needed values from the current state of the program, and all of these state functions will be encountered in the following contents. Components of specific states will be denoted using the appropriate Greek letter subscripted by the state of interest.

First of all, because all Lolisa formal specifications are constructed on GERM framework, the context of the formal memory space is denoted as \( M \), the context of the execution environment is represented as \( E \) and we assign \( A \) to denote the set of memory addresses and employ the meta-variable \( \sigma \) to represent an arbitrary address. Specially, the function return address \( A_{\text{fun}} \), which, in the current version of Lolisa, is assumed to be the next address after a function identifier. In addition, struct is a type of important data structure in Lolisa. Therefore, \( \Sigma \) represents the Lolisa struct information context, and \( \Theta \) is employed to represent the set of pointers of struct types. Besides, in the following type judgements may include variables, so our type will include references to variable-typing contexts, which we will denote as \( \Gamma \), \( I \), etc. Such contexts are finite mappings from variable names to types. Besides, we assign \( \Omega \) as the native value set of the basic logic system. And programs may also contain references to the Solidity program’s declared functions of a Solidity program, so that another map is needed, this mapping from function identifiers to types. For brevity, we will be written \( \Phi \), \( \Phi_1 \) etc. For shorthand, we will assign \( F \) to represent the formal system combination of \( \Sigma \), \( \Gamma \), \( \Theta \), \( \Omega \), \( \Phi \), and \( A \). Because they. All these terms represent different structure information of Lolisa different part infer the formal system world, and of Lolisa that will be needed in the following content, they will be needed all discussion.

| \( M \)  | Memory space |
|---------|-------------|
| \( A \)  | Memory address set |
| \( \Sigma \) | Struct information |
| \( \Gamma \) | Context structure information |
| \( \Phi \) | Function information |

\( A_{\text{fun}} \) Function return address
\( \Theta \) Struct pointer set
\( \Omega \) Native value set
\( F \) Formal system world

Table 1. State functions

2.2 Modification of the GERM framework

As discussed, the dynamic semantics of Lolisa are designed based on our GERM framework, which is also employed to represent or generate the intermediate memory state. We write \( \sigma \) to denote the specific memory state. The level of the formal memory space simulates a real-world physical memory structure, and consists of formal memory blocks used to store information, and the formal memory addresses used to index the respective memory blocks. Because of the formal memory space definition employed in the GERM framework, we can define special memory addresses to index special memory blocks isolated from the normal memory block. We define the formal memory space architecture by enumerating the memory blocks using the Record type to simulate the physical memory space intuitively in the higher order logic system of Coq. In the formal specification based on these useful features of the Record type, the memory address denoted by \( \text{address} \) is the field identifier of record type \( \text{memory} \), and each field can record a term denoted as \( \text{value} \). Each formal memory block can be abstracted as a Cartesian product \( (m_{\text{addr}}, \text{value}) \): \( \text{address} \times \text{value} \), where the metavariable \( m_{\text{addr}} \) is an arbitrary memory address, and the metavariable \( \text{value} \) is the value term stored in \( m_{\text{addr}} \), which includes the input data, and respective data environment and memory block state information. In the original version of the proposed framework, \( \text{value} \) can record 11 basic datatypes, including undefined, machine integer, Boolean, floating point, string, and array datatypes, in addition to pointers for variables, parameters, and functions, as well as program statement and struct datatypes. The level of low-level memory management operations analyzes requests for high-level memory management operations, and interacts with the formal memory space to generate the resulting memory state for those operation requests. Finally, the operation requests are executed at this level. The interactions on this level involve distinct low-level operations for normal memory blocks and special memory blocks. In the low-level memory management layer, the label address type is a transitional type in Coq that is employed to provide a simple memory address identifier for operation functions, and to isolate the low-level formal memory space from high-level formal specifications. We generally adopt the metavariable \( L_{\text{address}} \) to represent a label address. However, to support the syntax and semantics of Lolisa, we must modify and extend the GERM framework. The modification includes: 1) defining special label addresses and respective memory blocks for the built-in data structures and functions of EVM; 2) specifying abstract definitions of the environment and memory block states; 3) extending the range of types of \( m_{\text{value}} \).

First, Lolisa operates at the high-level specification in the GERM framework, such that the low-level memory address is transparent for Lolisa, and it can only operate label address. Therefore, for simplicity, the label address will be denoted as memory address or address in the following discussion. Each identifier of Lolisa programs (including identifiers for variables, parameters, struct types, functions, and contracts) is bound by an address to index the respective memory block in the GERM framework. For example, in Coq, an identifier is defined as
Definition identifier₁ := a₁.

And in the formal system of Coq or similar proof assistants, identifier₁ and a₁ are equivalent logic symbol, which can be abstracted as rule ID-ADDR. Therefore, in the following contents, it can be implied that each identifier represents a memory address.

identifier ≡ a₁ (ID-ADDR)

However, Solidity employs some built-in structures and functions in addition to the structures designed by users, and we assume in Lolisa that these built-in components are correct and trusted. These data structures and functions are defined and implemented using Lolisa syntax in advance, and we package them as a standard library, which will be discussed in detail in a later section. Obviously, these must also be stored in memory blocks indexed by memory addresses. To avoid overwriting these standard data during the execution (verification) of formal programs in theorem-proving assistants, we add some special addresses and respective memory blocks in the GERM framework to isolate these standard data structures from user data. Currently, we support the built-in data structures and functions send (also denoted as transfer in current EVM), call, msg, address, and block. Their addresses are defined as follows.

Special address: `_standard` ::= `_0xinit` | `_0xsend` | `_0xsend_re` | `_0xcall` | `_0xmsg` | `_0xaddress` | `_0xblock`

In addition, we introduce some special functions in the following sections that need no special addresses, such as requires [21].

Second, we specify the abstract definitions of the execution environment and memory block states, which are used to store the logic information of the native formal system, and are provided to the dynamic semantics of Lolisa. The abstract definitions of the environment and memory block states are given according to Solidity specifications below as Redefinitions 1 and 2.

Redefinition 1 (execution environment; env) In the current framework for Lolisa, the execution environment (denoted as env) includes four components:

- `inhers`: a list of terms with `L_address` type to store inheritance relationships between different contracts;
- `dom_super`: terms with `L_address` type to store the super-scope of the current scope;
- `dom_current`: terms with `L_address` type to store the current scope;
- `dom_level`: terms with `N` type to store the level of the current scope, where levels 0, 1, and 2 represent a local variable, a contract member variable, and a global variable, respectively;
- `gas`: represents the remaining amount of gas [21] in the current execution environment.

The inductive definition of environment follows the rule ENV-IND given below.

\[
\text{environment} \equiv \text{Inductive} (\text{inhers}, \text{dom}_{\text{super}}, \text{dom}_{\text{current}}, \text{dom}_{\text{level}}, \text{gas}) \quad \text{(ENV-IND)}
\]

We formally define `env` according to the rule ENV-TERM below. Here, `env` represents an arbitrary term constructed by only a single constructor `Env` according to the rule CST-ENV below.

\[
\begin{align*}
\text{env} : \text{Env} & \quad \text{(ENV-TERM)} \\
\alpha \vdash \text{inhers}, \text{list L_address} & \\
\alpha \vdash \text{dom}_{\text{super}}, \text{dom}_{\text{current}}, \text{option L_address} & \\
\alpha \vdash \text{dom}_{\text{level}}, \text{N} & \\
\alpha \vdash \text{gas}, \text{N} & \\
\alpha \vdash \text{Env}(\text{inhers}, \text{dom}_{\text{super}}, \text{dom}_{\text{current}}, \text{dom}_{\text{level}}, \text{gas}) : \text{Env}
\end{align*}
\]

(CST-ENV)

Redefinition 2 (memory block state; `b_infor`) The memory block state (abbreviated as `b_infor`) is employed to formalize the execution information of a memory block. In the current framework, `b_infor` includes the following memory occupancy, access authority, and type annotation definitions.

- `alloc`: This represents whether a current memory block is presently allocated. In Coq formalization, it is defined as an inductive type, and has two representations:

  \[
  \text{occupy} \mid \text{unoccupy}.
  \]

- `access`: This refers to the access modifier used to represent the access authority of the current memory block. In Coq formalization, it is defined as an inductive type, and has three representations:

  \[
  \text{public} \mid \text{protected} \mid \text{private}.
  \]

- `types`: The GERM framework employs a strong type memory space. Therefore, if a memory block is allocated to a identifier having a type \( \tau \), the memory block will be initialized by the respective initial memory value that satisfies the mapping relation with type \( \tau \). As such, `b_infor` will store a Lolisa type annotation that represents an expected type of the respective memory block indexed by the current identifier. This is defined in detail in the following section.

We formally define `b_infor` according to the rule BLC-TERM below. Here, `b_infor` represents an arbitrary term constructed by only a single constructor `Blc` according to the rule CST-BLC below. In addition, `b_infor` is contained within `M`.

\[
\begin{align*}
b_{\text{infor}} : \text{Blc} & \quad \text{(BLC-TERM)} \\
\Sigma \emptyset \vdash \text{alloc} & \\
\Sigma \emptyset \vdash \text{access} & \\
\Sigma \emptyset \vdash \text{type} & \\
\Sigma \emptyset \vdash \text{Blc} & \\
\Sigma \emptyset \vdash \text{Blc}(\text{alloc}, \text{access}, \text{type}) : \text{Blc}
\end{align*}
\]

(CST-BLC)
Finally, Solidity employs some typical values, such as mapping values, that are not supported by the general GERM framework. Therefore, we add the five new value definitions shown in Figure 1, which include contract values (VALUE-CONTR), byte values (VALUE-BYT), mapping values (VALUE-MAP), multiple-function-return values (VALUE-TYPES), and send return values (VALUE-SEND). In addition, we have defined special memory values denoted as \( \text{value}_{map} \) to correspond to the memory values \( \text{type}_{map} \) and \( \text{val}_{map} \) of Lolisa, which will be described in detail in the following sections. Because \( \text{value}_{map} \) is a subset of \( \text{value} \) that does not include the typing rules VALUE-MAP and VALUE-SEND, \( \text{value}_{map} \) is defined by rule MAP-V below, where the symbol \( \sqsubseteq \) is adopted to represent deleting rather than listing its rules redundantly.

\[
\text{value}_{map} \sqsubseteq \text{value} \quad \text{(VALUE-MAP, VALUE-SEND)} \quad \text{(MAP-V)}
\]

In Coq formalization, \( \text{value}_{map} \) is defined as a new type whose constructors are prefixed by \( i \).

### 3. Formal Syntax of Lolisa

Lolisa is a large formal subset of Solidity that is structured into type annotations, values, expressions, statements, functions, and modules, and its syntax is structured into five levels that the type level, value level, expression level, statement level and module level. In Coq formalization, the formal abstract syntax is presented as inductive predicates, and therefore achieves a deep embedding of Lolisa in Coq. The formal abstract syntax tree of Lolisa is nearly equivalent to that of Solidity, in that it contains nearly all the components of the original Solidity syntax. However, owing to the stated goal of Lolisa, it also includes some modifications to ensure more effective program verification. One of the differences between Solidity and Lolisa is that the Lolisa syntax of values and expressions are redefined as strongly typed using GADTs, which allows us to define types of syntax constructors directly. As discussed, it is therefore impossible to construct ill-typed terms, and the formal syntax ensures that all expressions and values in Lolisa are deterministic. Moreover, the formal static and dynamic semantics are more easily defined and understood, and the evaluator for these semantics is easier to read and write, as well as being more efficient. In addition, the type annotations not only transmit the type information, but can also store the value information, which will be discussed in detail in the following subsections. Other differences are that the current version of Lolisa omits some unnecessary qualifiers, does not support the inline assembly of Solidity, and only \( wei \) units are supported implicitly.

#### 3.1 Types

The formal abstract syntax of Lolisa types is given in Figure 2. Supported types include arithmetic types (integers in various sizes and signedness), byte types, array types, mapping types, as well as function types and struct types. Although Solidity is a JavaScript-like language, it supports pointer reference. Therefore, Lolisa also includes pointer types (including pointers to functions) based on label address specification. Furthermore, these type annotations and relevant components can be easily formalized by enumerating inductively in Coq or other higher-order logic theorem-proving assistants. Lolisa does not support any of the type qualifiers such as const, volatile, and restrict, and these qualifiers are simply erased during parsing.
The types fill two roles in Lolisa. Firstly, they serve as type declarations of identifiers, and, secondly, they serve as tags to specify the constructor types of values and expressions. In Coq formalization, the term \( \tau \) is declared as type \( \text{type} \) according to rule \( \text{TYPE-TERM} \) below.

\[
\text{type} \quad (\text{TYPE-TERM})
\]

Note that many \( \text{types} \) are defined in Figure 2 as parameterized types recursively. In this way, a specific \( \text{type} \) is dependent on the specified parameters, and can be abstracted and expressed in many different Solidity types.

One of the most important data types of Solidity is mapping types. In Solidity documentation [21], mapping types are declared as mapping (_KeyType \( \Rightarrow \) _ValueType). Here, _KeyType can be nearly any type except for a mapping, a dynamically sized array, a contract, and a struct. As shown in Figure 2, it is defined as \( \text{Map}(\tau_{\text{map}}, \tau) \), where \( \tau_{\text{map}} \) represents the _KeyType and \( \tau \) represents the _ValueType. The best way to keep the terms in Lolisa well-typed and to ensure type safety is to maintain type isolation rather than adding corollary conditions. Therefore, we define a coordinate type \( \text{type}_{\text{map}} \) for _KeyType employed in mapping. In particular, the address types in Lolisa are treated as a special struct type, so that _KeyType is allowed to be a struct type in Lolisa. In Coq formalization, \( \text{type}_{\text{map}} \) shares the same constructor with that of \( \text{type} \) except for \( \text{Map} \), and a term with type \( \text{type}_{\text{map}} \) is recorded as \( \tau_{\text{map}} \) according to the rule \( \text{MAP-TYPE-TERM} \) below.

\[
\tau_{\text{map}} \quad \text{type}_{\text{map}} \quad (\text{MAP-TYPE-TERM})
\]

In Solidity, array types, which defined as \( \text{Array}(id_{\text{array}}, \tau) \) in Coq, where \( id_{\text{array}} \) represent the array index, can be classified as fixed size arrays and dynamic size arrays. For fixed size arrays, the size and index number are allowed to be declared by different identifiers including constants, variables, struct, mapping, and field access values, defined as \( \text{Aconst}_{\text{id}}(n), \text{Avar}_{\text{id}}(\alpha), \text{Astr}_{\text{id}}(\alpha, \text{mems}), \text{Amap}_{\text{id}}(\alpha, \text{id}_{\text{maparray}}) \) and \( \text{Array}_{\text{id}}(\alpha, \text{id}_{\text{array}}) \) respectively, where \( \alpha \) represents an identifier indexed by an address, and \( \text{mems} \) represents the member name of structure members. Specially, because the index value \( \text{Amap}_{\text{id}} \) is forbidden to employ itself recursively, \( \text{id}_{\text{maparray}} \) is the coordinate index defined in Figure 2 for mapping values, which shares the same constructor with that of \( id_{\text{array}} \) except for \( \text{Amap}_{\text{id}} \). Therefore, array types of \( \tau \) are parameterized in Lolisa by \( id_{\text{array}} \), while array types of \( \tau_{\text{map}} \) are parameterized by \( id_{\text{maparray}} \), which does not include a mapping identifier. As shown in Figure 2, \( id_{\text{array}} \) and \( id_{\text{maparray}} \) contain the respective formal constructors to represent different index forms. Here, we assume that all indices with memory addresses in the set \( A \) are well-formed. The formalizations \( id_{\text{array}} \) and \( id_{\text{maparray}} \) not only can refer to the size of respective arrays, but can also be used to refer to the index number during evaluation. For example, in Lolisa, on the one hand, when declaring an identifier \( A \) defined by type EXP-ARR as EXP-ARR-VAR, EXP-ARR-VAR signifies that the identifier \( A \) represents an array of type \( \tau \) with a size of 10. On the other hand, as EXP-ARR-ASSIGN, if EXP-ARR signs a right value \( B \) of assignment, it signifies that the left value \( L \) in this assignment statement will be replaced by the 10th element of array \( B \). (The \text{Var}, \text{Assign} and \text{Evar} will be explained in the following sections.)

\[
\text{Array} (\text{Aconst}_{\text{id}}(10), \tau) \quad (\text{EXP-ARR})
\]

\[\text{Var} \left( \text{Evar} A \left( \text{Array} (\text{Aconst}_{\text{id}}(10), \tau) \right) \right) \quad (\text{EXP-ARR-VAR})\]

\[\text{Assign} L \left( \text{Evar} B \left( \text{Array} (\text{Aconst}_{\text{id}}(10), \tau) \right) \right) \quad (\text{EXP-ARR-ASSIGN})\]

Because the size of array types in Solidity can be dynamic, the dynamic size array type in Lolisa is treated as a special mapping type of \( \tau_{\text{map}} \) (Int Signed \( 64 \)). In addition, array types and mapping types are defined recursively. Due to the recursive inductive definition, Lolisa can express \( n \)-dimensional array types and \( n \)-dimensional mapping types easily, which is illustrated below by examples EXP-ARR-N and EXP-MAP-N, respectively.
Type level Declare type annotation
Value level Represent struct value and field value
Expression level Represent right value of struct type
Statement level Declare new struct type

Convention 1. Formal struct datatype

Lolisa includes four kinds of pointer types: variable pointers, parameter pointers, function pointers, and contract pointers, which are defined as dependent types formally. Thus, each type is specified by a specific optional memory address \( \alpha \) with the type \( \text{optional L-address} \), which is used to express the situation of the NULL pointer. Specifically, if a reference identifier points to a NULL pointer or fails to allocate a logic memory address, \( \alpha \) is None. Otherwise, \( \alpha \) is Some \( \alpha \). However, the NULL pointer is invalid in the semantics for Solidity during evaluation. During evaluation, a formal interpreter or compiler can index a respective memory block by \( \alpha \) directly, and this design is used for extension which will be explained in Section 7 NULL NULL.

In subsequent discussion, we employ the construction \( \Sigma, \theta, A \vdash wf(\tau) \) to represent that type \( \tau \) is well-formed in the Lolisa struct information context \( \Sigma \), assuming that all pointers to struct types with tags in the set \( \theta \) are also well-formed. In addition, types may require a label address, so the parameter \( A \) is necessary. Furthermore, the construction \( \Sigma \vdash wf(\tau) \) represents that \( \theta \) and \( A \) are equal to the empty set, which will be our typical meaning when employing the term well-formed. First, the normal form types are well-formed:

\[
\Sigma, A \vdash wf(\tau_nf) \quad \text{(WF-NORM)} \quad \Sigma, A \vdash wf(\tau_{nf/map}) \quad \text{(WF-NORM-MAP)}.
\]
Second, array types are well-formed if their normal form type is well-formed and not void ($\mathcal{T}undef$), and if the number of elements is greater than zero. We write $\downarrow_{idarray}$ and $\downarrow_{idmaparray}$ to represent evaluations of array indices, and their type assessments are respectively defined as rules WF-ARR and WF-ARR-MAP below.

$$\frac{\Sigma, \theta, A \vdash \text{wf}(\tau)}{M, \Sigma, \theta \vdash \text{wf}(\tau[\downarrow_{idarray}])} \quad \text{(WF-ARR)}$$

$$\frac{\Sigma, \theta, A \vdash \text{wf}(\tau_{map}) \quad M, \Sigma, \theta \vdash \text{wf}(\downarrow_{idmaparray}[\downarrow_{idmaparray}])}{M, \Sigma, \theta \vdash \text{wf}(\tau_{map}[\downarrow_{idmaparray}])} \quad \text{(WF-ARR-MAP)}$$

Finally, mapping types are well-formed only if their arguments $\_\text{KeyType}_{\tau_{map}}$ and $\_\text{ValueType}_{\tau}$ are well-formed, which is expressed by the rule WF-MAP below.

$$\frac{\Sigma, \theta, A \vdash \text{wf}(\tau) \wedge \text{wf}(\tau_{map})}{\Sigma, \theta, A \vdash \text{wf}(\text{mapping}[\tau_{map}, \tau])} \quad \text{(WF-MAP)}$$

Henceforth, our meta-variables $\tau$, $\tau_1$, $\tau_2$ etc. will range only over well-formed types.

### 3.2 Values

Most similar formalization studies focused on high-level programming languages do not contain a value-level formalization. The present motivation for defining a value level for Lolisa is that the target of this project is to formalize mechanized syntax and semantics for a subset of the Solidity language that can be executed and verified in higher-order logic theorem-proving assistants directly. Therefore, Solidity values must be evaluated like the native values in the formal system. The best possible situation would be to employ the values of Solidity or those of some mainstream high-level programming language explicitly in the formal system. However, due to the strict typing system of the trusted core and the adoption of different paradigms, nearly all higher-order logic theorem-proving assistants, such as Gallina (Coq), do not directly support complex values, such as array values and mapping values. Therefore, it is essential to define an interlayer between the values of real-world languages and the native values of the formal system that can represent the real world-values directly with an equivalent syntax, and can translate the real world-values into the native values using formal semantics.

| Variable address: $\text{varid := vid}(a)$ | Parameter address: $\text{parid := pid}(a)$ |
| --- | --- |
| Function address: $\text{funid := fid}(a)$ | Contract address: $\text{conid := cid}(a)$ |
| Field type list: $\text{fitype := Fstruct}(a, \ldots) \mid \text{Fmap}(a, \ldots) \mid \text{Farray}(a, \ldots)$ | Field parameter: $\text{strn} := \text{Nvar}(\text{name}) \mid \text{Narray}(\text{name}, a, \text{index})$ |
| Map Index $\text{id}_{\text{map}} := \text{Mconst}_{\text{id}}(\alpha) \mid \text{Mvar}_{\text{id}}(a) \mid \text{Mstr}_{\text{id}}(a, \text{mem}) \mid \text{Marray}_{\text{id}}(a, \text{id}_{\text{map}}) \mid \text{Map}_{\text{id}}(\text{id}_{\text{map}}, \text{id}_{\text{map}})$ |

| Lolisa normal form field value $\text{iv}_{\text{map}} := \text{Vundef}(t) \mid \text{Vint}(t) \mid \text{Vbool}(b) \mid \text{Vbyte}(b) \mid \text{Vstring}(s) \mid \text{Vstruct}(\text{addr}, \text{addr}')$ |
| --- |
| $\text{iv}_{\text{map}}(\text{varid}) \mid \text{iv}_{\text{map}}(\text{parid}) \mid \text{iv}_{\text{map}}(\text{funid}) \mid \text{iv}_{\text{map}}(\text{conid})$ |
| Lolisa field value: $\text{ival} := \text{Narray}(\text{id}_{\text{array}}, \tau, \text{addr}) \mid \text{Nmap}(\text{id}_{\text{map}}, \text{id}_{\text{map}}, \text{ov})$ |
| $\text{ival}_{\text{map}}(\text{varid}) \mid \text{ival}_{\text{map}}(\text{parid}) \mid \text{ival}_{\text{map}}(\text{funid}) \mid \text{ival}_{\text{map}}(\text{conid})$ |

| Lolisa normal form value $\text{val}_{\text{map}} := \text{Vundef}(t) \mid \text{Vint}(t) \mid \text{Vbool}(b) \mid \text{Vbyte}(b) \mid \text{Vstring}(s) \mid \text{Vstruct}(\text{addr}, \text{addr}')$ |
| --- |
| $\text{val}_{\text{map}}(\text{varid}) \mid \text{val}_{\text{map}}(\text{parid}) \mid \text{val}_{\text{map}}(\text{funid}) \mid \text{val}_{\text{map}}(\text{conid})$ |

| Lolisa value: $\text{val} := \text{Narray}(\text{id}_{\text{array}}, \tau, \text{addr}) \mid \text{Nmap}(\text{id}_{\text{map}}, \text{id}_{\text{map}}, \text{ov})$ |
| --- |
| $\text{val}_{\text{map}}(\text{varid}) \mid \text{val}_{\text{map}}(\text{parid}) \mid \text{val}_{\text{map}}(\text{funid}) \mid \text{val}_{\text{map}}(\text{conid})$ |

Figure 3. Abstract syntax of Lolisa values

The syntax of values employed in Lolisa is given in Figure 3. All values are signed by the types defined in the previous subsection. Actually, value is a type dependent on the specific type signature, as given by rule LOS-VAL below.

$$\text{val} : \text{type} \rightarrow \text{Type} \quad \text{(LOS-VAL)}$$

In this way, all values in Lolisa are well-typed, such that it is impossible to construct an ill-typed value such as $\text{Vint}(\text{true})$. In addition, the type information is transmitted to the expression level for maintaining well-typed values. For example, a value $v$ has type $\text{val} \tau_1$, and the constant expression $\text{Econst}$, which will be defined in Section 3.3, has type $\forall (\tau : \text{type})$. $\text{val} \tau \rightarrow \text{expr} \downarrow_{\tau_1} \downarrow_{\tau_1}$. And the $\tau$ in $\text{Econst}(v)$ is determined by $\tau_1$. In this way, the type information of value level is transmitted to expression level. The more specific details will be explained in the next section. We assign $v$ to represent terms with type $(\forall \tau : \text{type}, \text{val} \tau)$, as given by the rule LOS-VAL-TERM below.

$$v : (\forall \tau : \text{type}, \text{val} \tau) \quad \text{(LOS-VAL-TERM)}$$

Another important component of the values in Lolisa is the parallel mapping value $\text{value}_{\text{map}}$, which corresponds to $\text{type}_{\text{map}}$. In this way, type safety and type isolation can be transmitted from the type level to the current value level. It should be noted that we do not discriminate
between values and mapping values because the syntax of mapping values is nearly equivalent to that of values which is defined in Figure 3. The only differences between their respective syntaxes are that mapping values are annotated by type\text{map} and mappings are not defined recursively. These differences do not affect the value-level formalization.

As was employed for types, the value formalizations are also classified as normal and non-normal forms. But different from the classification principle of normal and non-normal form types, the normal form value val\text{nf} signifies that a value can access the native value of the formal system directly, rather than searching and matching values stored in the memory space, and that val\text{nf} types have a one-to-one correspondence with \(\tau\text{nf}\) types. Moreover, val\text{nf} types can be easily formalized using inductive enumeration. For example, the rule EXM-VAL-BOOL below presents the type rule of Boolean values, while the remaining val\text{nf} type rules are defined in an equivalent manner.

\[ \Omega \vdash \lambda : \text{Bool}. \ \text{Vbool} \ : \ \text{val} \text{Tbool} \quad (\text{EXM-VAL-BOOL}). \]

As such, values are obviously well-formed types, and are written as \(\text{wf(val}\text{nf})\). Non-normal form values include the values of arrays, mappings, and field access. Essentially, array and mapping values can be obtained by evaluating their indices. As shown in EXP-ARR above, \(id\text{array}\) not only can refer to the size of an array, but can also be used as the key of array elements. Therefore, array values in Lolisa can be constructed according to the rule VAL-ARR below.

\[ \begin{align*}
\Sigma, \theta, \lambda \vdash \text{type} & \quad \Lambda \vdash \text{name: address} \quad M \vdash \text{index:id array} \\
\Sigma, \theta, \lambda \vdash \text{Vararray}(\text{v(id:id array)}(\text{type})\text{address}=\text{val}(\text{Vararray}(\text{index} \ i)) \quad (\text{VAL-ARR})
\end{align*} \]

The mapping formal rule is defined by VAL-MAP below.

\[ \begin{align*}
\Sigma, \theta, \lambda \vdash \text{t0:typemap} & \quad \Sigma, \theta \vdash \text{t1: type} \quad M \vdash \text{key: id map} \ t0 \\
\Sigma, \theta, \lambda \vdash \text{snd: option(val t1) \ & (snd=None \ \lor \ \text{snd=Some}(\text{vmap}(\text{t})))} \\
\Sigma, \theta, \lambda, \rho \vdash \text{vmap}(\text{v(t0:typemap)}(\text{t1: type})\text{address=index map} \ t0=\text{option(val t1) =val(\text{Vmap t0 t1})} \\
\Sigma, \theta, \lambda, \rho \vdash \text{w(Varmap(head, key, snd))} = \text{val(Varmap t0 t1)} \\
\end{align*} \]

(VAL-MAP)

While the keys of mapping values are allowed in Solidity to be represented by different identifiers including constants, variables, structs, arrays and mappings values, here, like \(id\text{array}\), we have formalized the key of mapping values as \(id\text{map}\), and its constructors are given in Figure 3. The \(\text{snd}\) parameter is used to indicate the next dimensionality of mapping. If \(\text{snd}\) is \(\text{None}\), it indicates that \(\text{snd}\) does not include the next dimensionality. Otherwise, if \(\text{snd}\) is \(\text{Some}(\text{Vmap(*))}\), where the symbol * is a wildcard, we should search for the next dimensionality. Typically, field access is defined as a value rather than as an expression or statement. This is the case in Solidity, which allows both field access or method access to be defined as values. An example of this is shown by the Solidity code segment given in Figure 4 [22], where the members of the struct \(\text{pledge[i]}\) are invoked as values, which is common usage in Solidity. Therefore, field access is formalized in Lolisa at the value level.

The formal type assessment of field access is defined as the rule VAL-FIELD below.

\[ \begin{align*}
\Sigma, \theta, \lambda \vdash \text{t1: type} & \quad \Sigma, \theta \vdash \text{head: field\text{type}} \ t0 \\
\Sigma, \theta \vdash \text{mems: list str name} \\
\Sigma, \theta, \lambda \vdash \text{opars: option(list ival)} \ & \land \text{w(f(opars))} \\
\Sigma, \theta, \lambda, \rho \vdash \text{vfield}(\text{v(t1: type)}\text{field\text{type}} \ t0=\text{list str name}=\text{option(list ival)}=\text{val(t1)} \\
\Sigma, \theta, \lambda, \rho \vdash \text{w(Vfield(head, mems, opars))} \\
\end{align*} \]

(VAL-FIELD)

\[ \begin{align*}
\text{function refund} () \ {\}
\quad \text{if (msg.sender != owner || complete || refunded);} \ {\}
\quad \text{throw;} \\
\quad \text{for (uint i = 0; i < numPledges; i++)} \ {\}
\quad \text{pledges[i].eth Address.send(pledges[i].amount);} \\
\quad \text{refunded} = \text{true}; \\
\quad \text{complete} = \text{true}; \\
\end{align*} \]

(4) Smart sponsor contract code [22]

Note that, if the final member of the member list is a function call, we employ the list \(ival\) to transmit the arguments. Here, \(ival\), defined in Figure 3, is similar to \(val\), except it does not include type annotation. It is unnecessary to limit the types of arguments that will be checked by the helper function in dynamic semantics. Finally, the rule VAL-STR below defines the formal type assessment of struct.

\[ \begin{align*}
\Lambda \vdash \text{str\text{type}: address} \\
\Lambda \vdash \text{Vstruct}(\text{v str\text{type}: address=address} = \text{val}(\text{vstruct str\text{type}})) \\
\Lambda \vdash \text{w(Vstruct(str\text{type}, str\text{type}))} \\
\end{align*} \]

(VAL-STR)

As given by Convention 1, struct is also treated in Lolisa as values, which is used to represent an expression value at the right position with a struct type stored in memory space.

Following the above definitions, we can assume that the values in Lolisa are well-formed, and our meta-variables \(v, v_1, v_2\), etc. will range only over values satisfying \(\text{wf}(v)\).
### 3.3 Expressions

Having formally specified all the possible forms of values that may be declared and manipulated in Solidity programs, we now discuss the expressions used in programs to wrap values. The formal syntax of expressions is given in Figure 5. All expressions and their sub-expressions are annotated by two type signatures according to the rule EXPR-TYPE below:

\[ \text{expr} : \tau_0 \rightarrow \tau_1 \rightarrow \text{Type} \ (\text{EXPR-TYPE}) \]

Here, \( \tau_0 \) refers to the current expression type and \( \tau_1 \) refers to the normal form type after evaluation. For instance, we would define the type of an integer variable expression \( e \) as \( \text{expr} \tau_{\text{int}}(\alpha) \rightarrow \tau_{\text{int}} \). In this way, the formal syntax of expressions becomes more clear and abstract, and allows the type safety of Lolisa expressions to be maintained strictly. In addition, employing the combination of the two type annotations facilitates the definition of a very large number of different expressions based on equivalent constructors. Of course, the use of \( \tau_0 \) and \( \tau_1 \) may be subject to different limitations depending on the situation.

Within expressions, Lolisa does not support assignment operators (=, +=, ++, etc.) and function calls, and otherwise supports only Solidity operators that are free of side effects during evaluation. In Lolisa, assignments and function calls are presented as statements, and cannot occur within expressions. In addition, unary assignment operators, such as increment or decrement operators, are treated as syntactic sugar. As a consequence, all Lolisa expressions always terminate during evaluation and are pure, i.e., their evaluation incurs no side effects. In this way, syntactic expressions can be safely used as components in logical assertions, such that it is much easier to define axiomatic semantics such as Hoare logic and separation logic. Likewise, abstract interpretations and other forms of static analysis are greatly simplified if expressions are pure because most static analysis and program verification tools begin by removing assignments and function calls from expressions, and only then perform analyses over the resulting pure expressions [23]. The expressions of Lolisa are classified according to four categories, including constant expressions, location reference expressions, special expressions, and operator expressions, all of which are discussed in detail as follows.

![Figure 5. Abstract syntax of Lolisa expressions](image)

Constant expressions are used to denote the native values of the basic formal system, which are transformed from respective Lolisa values. Therefore, \( \tau_0 \) and \( \tau_1 \) should satisfy the limitation TYPE-FORM given below.

\[ \tau_0 = \tau_1 \wedge \tau_0, \tau_1 \in \tau_{nf} \ (\text{TYPE-FORM}) \]

To satisfy the limitation TYPE-FORM, the array types and mapping types should be analyzed and simplified according to the type definitions given by Figure 2 into \( \tau_{\text{final}} \in \tau_{nf} \), which can be formulated as \( \Sigma, \Theta \vdash \tau \rightarrow \tau' \rightarrow \cdots \rightarrow \tau^n \wedge \tau^n \in \tau_{nf} \). We denote this process as \( \downarrow \). Therefore, the type assignment of constant expressions is defined as the rule EXPR-CONS below.

\[ \Sigma, \theta, \lambda \vdash \tau : \text{type} \quad \alpha \vdash v : \tau \quad \tau_{nf}() \quad \mathcal{P} / \Phi = \text{Econs}(\tau : \text{val}) , \text{Eapprt}(\tau : \text{vals}), \text{Eapprf}(\tau : \text{fun}) \]

\[ \mathcal{P} / \mathcal{W} / \mathcal{E} / \text{Efun}(\tau : \text{fun}) \] \quad \text{EXPR-CONS} \]

Location reference expressions include the identifiers of variables, parameters, functions, and contracts, which are summarized as \( eaddr \) in Figure 5. For the type assignment of location reference expressions, the types of constructors are defined as functors. For example, \( \text{Evar} \) has the type defined by FUNCTOR-VID below.

\[ \lambda (\alpha : \text{option}(\lambda \text{addr})) \rightarrow (\text{expr } (T \text{vid } \alpha) \rightarrow \tau) \ (\text{FUNCTOR-VID}) \]

Here, the use of an optional address (the variable identifier) and another type as parameters is facilitated under GADT style definition to denote a specific type. Note that the address is optional because it includes conditions requiring the NULL pointer for representing an expendable space for expending Lolisa to support more general-purpose programming languages, although the NULL pointer is invalid in current Lolisa semantics for Solidity. Therefore, the type assignment of location reference expressions is defined as the rules EXPR-PAR, EXPR-VAR, EXPR-FUN, and EXPR-CON below for the identifiers of parameters, functions, variables, and contracts, respectively.
Special expressions include struct expressions \( \text{Extract} \) and modifier expressions \( \text{Emodifier} \). First, at the expression level given in Convention 1, the only function of \( \text{Extract} \) is to represent an expression value at the right position, which is used to initialize or modify struct type terms. Therefore, its rule type has two parameters: the respective struct type identifier and the value list \( \text{str}_{\text{par}} \) for each member field, as defined by the rule \( \text{EXPR-STR} \) below.

\[
\Sigma, \Theta, \Lambda \vdash \tau : \text{type} \quad \Lambda \vdash \alpha : \text{option Laddress} \\
F / \Phi \vdash \text{Expr} : (\tau : \text{type}) (\text{addr} : \text{option Laddress}) \text{expr} (\text{type} \text{addr}) r \tau \\
\Sigma, \Theta, \Lambda \vdash \tau : \text{type} \quad \Lambda \vdash \alpha : \text{option Laddress} \\
F / \Phi \vdash \text{Efun} : (\tau : \text{type}) (\text{addr} : \text{option Laddress}) \text{expr} (\text{type} \text{addr}) r \tau \\
\Sigma, \Theta, \Lambda \vdash \tau : \text{type} \quad \Lambda \vdash \alpha : \text{option Laddress} \\
F / \Phi \vdash \text{Econ} : (\alpha : \text{option Laddress}) \text{expr} (\text{type} \text{addr}) (\text{type} \text{addr}) r \tau \\
\]  

(\text{EXPR-PAR})

(\text{EXPR-VAR})

(\text{EXPR-FUN})

(\text{EXPR-CON})

Here, the value list is defined as \( \text{str}_{\text{par}} \) because, while the formal syntax of Lolisa values is dependent on an arbitrary type \( \tau \), the list elements should be of equivalent types. Thus, we can define a wrapper in Coq formalization that cleverly hides the type \( \tau \). For \( \text{Emodifier} \) expressions, Solidity includes a modifier as a special kind of functor [21] that can limit the behavior of functions annotated by it. Therefore, the only function of \( \text{Emodifier} \) in Lolisa is to declare an identifier to a modifier, and thereby maintain type isolation from normal functions. It is constructed using \( \text{Efun} \) to represent the identifier, and employs the list \( \text{ival} \) to transmit its parameters, as given by the type rule \( \text{EXPR-MODI} \) below.

\[
\Sigma, \Theta, \Lambda \vdash \tau : \text{type} \quad \Lambda \vdash \alpha : \text{option Laddress} \\
\Sigma, \Theta, \Lambda \vdash \alpha : \text{option Laddress} \vdash \alpha : \text{option Laddress} \\
F / \Phi \vdash \text{Efun} : (\tau : \text{type}) (\text{addr} : \text{option Laddress}) \text{expr} (\text{type} \text{addr}) r \tau \\
\Sigma, \Theta, \Lambda \vdash \tau : \text{type} \quad \Lambda \vdash \alpha : \text{option Laddress} \\
F / \Phi \vdash \text{Econ} : (\alpha : \text{option Laddress}) \text{expr} (\text{type} \text{addr}) (\text{type} \text{addr}) r \tau \\
\]  

(\text{EXPR-STR})

(\text{EXPR-MODI})

For operator expressions, Lolisa supports nearly all binary and unary operators. In Coq formalization, binary and unary operators are abstracted as two inductive types \( \text{op}_2 \) and \( \text{op}_1 \) that are also defined by GADTs, and specific operators serve as their constructors. In this way, operator expressions are made more clear and concise, and can be extended more easily than when employing a weaker static type system. The binary and unary operators are annotated by two type signatures, as respectively given in rules \( \text{EXPR-BOP-TYP} \) and \( \text{EXPR-UOP-TYP} \) below.

\[\text{op}_2 : \tau_0 \rightarrow \tau_1 \rightarrow \text{Type} \quad (\text{EXPR-BOP-TYP})\]

\[\text{op}_1 : \tau_0 \rightarrow \tau_1 \rightarrow \text{Type} \quad (\text{EXPR-UOP-TYP})\]

As was presented for the standard rule \( \text{EXPR-TYP} \), \( \tau_0 \) refers to the operator input type and \( \tau_1 \) refers to the output type after evaluation. Therefore, \( \tau_0 \) and \( \tau_1 \) must satisfy \( \tau_0, \tau_1 \in \tau_{nf} \). For example, the conjunction operator for Boolean values is defined as \( \text{fandOfBool} : \text{op}_2 \text{Tbool Tbool} \). In addition, due to the requirement for type limitation, each operator belongs to an operator class. For example, we define distinct equivalent operators for integer values, Boolean values, and address values as \( \text{feqOfInt} : \text{op}_2 \text{Tint (signedness, intsize) Tbool} \), \( \text{feqOfBool} : \text{op}_2 \text{Tbool Tbool} \), and \( \text{feqOfAddr} : \text{op}_2 \text{Taddress Tbool} \), respectively. On the one hand, this method maintains strong type limitations for each operator, and ensures that the behavior of each operator is deterministic. As a result, clear rules are given to the compiler and interpreter. On the other hand, the method ensures that Lolisa operators can be extended easily by adding new operator constructors. The Coq formalization of Lolisa includes nearly 100 kinds of specific operation type constructors for \( \text{op}_2 \) and \( \text{op}_1 \). For the remainder of this paper, we adopt \( \text{op}_{\text{class}}(\text{operator}) \) to simplify the formal abstract syntax. Based on the operator formalizations, we can define the formal type rules of operator expressions according to rules \( \text{EXPR-UOP} \) and \( \text{EXPR-BOP} \) below.

\[
\Sigma \vdash \tau_0 \quad \tau_1 \quad \tau_2 : \text{type} \\
F / \Phi \vdash \text{Expr} : (\tau_0) \text{expr} (\tau_1) r \tau_2 \\
\Sigma \vdash \tau_0 \quad \tau_1 \quad \tau_2 : \text{type} \\
F / \Phi \vdash \text{Cons} : (\tau_0 \times \tau_1) \text{expr} (\tau_2) r \tau_2 \\
\Sigma \vdash \tau_0 \quad \tau_1 \quad \tau_2 : \text{type} \\
F / \Phi \vdash \text{Cons} : (\tau_0 \times \tau_1) \text{expr} (\tau_2) r \tau_2 \\
\]  

(\text{EXPR-OP})

(\text{EXPR-BOP})
That operator expressions are well-formed is guaranteed by the static typing rules, and it is impossible to construct ill-formed operations, such as “error” + 1, because ill-formed operator expressions cannot pass the GADT-style formal type-checking rules defined above. In particular, type casting is defined as a unary operator because it can be seen as transforming the type $\tau_1$ of expressions to a type $\tau_2$.

As discussed above, some forms of Solidity expressions are forbidden, but can be expressed as syntactic sugar, for example:

\[(+ +) \equiv \text{Ebob}(\text{op\_class}(+)) \text{e}_2, \text{Tint}(s, t) \left( \text{Econst}(\text{Vint\_int}(s, t, 1)) \right)\]

In the following discussion, our meta-variables $e, e_1, e_2$, etc. will range only over well-formed expressions satisfying $wf(e)$.

### 3.4 Statements

Figure 6 defines the syntax of Lolisa statements. Here, nearly all the structured control statements of Solidity (i.e., conditional statements, loops, structure declarations, modifier definitions, contracts, returns, multi-value returns, and function calls) are supported, but Lolisa does not support unstructured statements such as goto and unstructured switches like the infamous “Duft’s device” [29]. As previously discussed, the assignment $e_1 = e_2$ of a right-value ($r$-value) $e_2$ to a left-value ($l$-value) $e_1$, and modifier declarations, as well as function calls and structure declarations are treated as statements. In addition, statements are also classified according to normal form and non-normal form categories, where the normal form statement, given as $\text{stt}_{nf}$, represents a statement that halts after being evaluated. Actually, although Solidity is a Turing-complete language, smart contract programs written in Solidity have no existing halting problems because the programs are limited by gas [25], which we have defined in $E$ of Lolisa.

![Figure 6. Abstract syntax of Lolisa statements](image)

As defined in Figure 6, we still inductively classify the statement definition into normal form $\text{stt}_{nf}$, whose typing judgements without recursive definition, and non-normal form statements. The normal form statements of Lolisa include variable declaration ($\text{Var}$), structure declaration ($\text{Struct}$), assignment ($\text{Assign}$), return ($\text{Return}$), multi-values return ($\text{Returns}$), skip ($\text{Skip}$), function stop flag ($\text{Fstop}$) and throw ($\text{Throw}$). The remaining statements are non-normal form statements. The formal assignment rules of $\text{stt}_{nf}$, and the non-normal form conditional ($\text{If}$) and sequence ($\text{Seq}$) statements are very simply defined according to the following 9 rules.

The assignment rules for structure declarations ($\text{Struct}$) require special treatment, and are therefore discussed later in this subsection.

Owing to formal syntax system in Lolisa employing GADTs, Lolisa statements are all well-formed. For example, Solidity or other formal languages such as Clight [24] allow the formation of the following examples of erroneous syntax:
Although these syntax errors will be discovered during compilation, such errors can seriously affect the evaluation of programs in higher-order theorem-proving assistants. However, constructing ill-formed statements is not possible in Lolisa because the type annotations have been fixed in the Lolisa formal abstract syntax tree. For example, the \( r \)-value \( e_r \) and \( l \)-value \( e_l \) in the rule \( \text{STT-ASSIGN} \) must have equivalent final types, and the return type of the condition expression given by the rule \( \text{STT-IF} \) must be Boolean. In addition, the current statement \( s_\text{eq} \) in the rule \( \text{STT-SEQ} \) cannot be a sequence because such a condition would create a confusing program structure. Therefore, errors are discovered through static type checking, which ensures that all statements are well-formed. In addition, conditional and sequence statements are well-formed if their sub-statements are not function declarations and contract declarations. Therefore, we denote the sequence of sub-statements by \( \Sigma^{\text{seq}}(s) \).

The specification of a function in Solidity is given as follows [21]:

\[
\text{function } ((\text{parameter types})) \{ (\text{access}) \ [\text{constant}] \ [\text{payable}] \ [\text{returns ((return types))}] \}.
\]

where the square brackets ([]) indicate an optional component. This specification is too complex to maintain clarity using a single type rule.

Therefore, single-value-return functions (\text{Fun}) and multi-value-return functions (\text{Funs}) are defined separately to maintain clear and well-formed definitions, and we adopt the option type to indicate optional components. The variable \text{pars} represents a list of parameters that stores expressions whose current type must be (\text{Tmod} \text{oa}). Due to the strong typing provided by GADTs, the expression constructor \text{Epar} in the higher-order logic system of Coq is a type class that each specific parameter has different type. Therefore, \text{pars} is defined as a heterogeneous list that allows the elements stored in the \text{pars} to have different types. The quantifier \text{access} represents the functions come in three flavors: public, protected, and private. In addition, the return types of \text{Fun} statements are stored in an additional list. To ensure that such statements are well-formed, their bodies must not contain function and contract declarations. Note that anonymous functions are forbidden in Lolisa because all functions must have a binding identifier to ensure that they are well-formed. The formal typing rules of \text{Fun} and \text{Funs} are respectively defined according to the rules \text{STT-FUN} and \text{STT-FUNS} below.

\[
\Sigma, \Theta, \Lambda \vdash: \text{type } \Lambda \vdash: \text{option } \text{address } \alpha \vdash: \text{option access } \text{of:option flag } \text{option payable} \\
\Sigma, \Theta, \Lambda \vdash: \text{modislist } \text{expr } \text{Tmodi } \text{Tmodi} \vdash: \text{fpar:par} \vdash: \text{F} \vdash: \text{statement} \\
\text{STT-FUN}
\]

\[
\Sigma, \Theta, \Lambda \vdash: \text{type } \Lambda \vdash: \text{option } \text{address } \alpha \vdash: \text{option access } \text{of:option flag } \text{option payable} \\
\Sigma, \Theta, \Lambda \vdash: \text{modislist } \text{expr } \text{Tmodi } \text{Tmodi} \vdash: \text{fpar:par} \vdash: \text{F} \vdash: \text{statement} \\
\text{STT-FUNS}
\]

Modifiers in Lolisa are treated as a kind of special function, which declares a set of restricted conditions for limiting the invocation of normal functions. And the normal function invocation is thrown out if the checking of restricted conditions fails. The formal rule of modifiers is similar to that of function declarations, and is defined as the rule \text{STT-MODI} below.

\[
\Sigma, \Theta, \Lambda \vdash: \text{type } \Lambda \vdash: \text{option } \text{address } \alpha \vdash: \text{option access } \text{of:option flag } \text{option payable} \\
\Sigma, \Theta, \Lambda \vdash: \text{modislist } \text{expr } \text{Tmodi } \text{Tmodi} \vdash: \text{fpar:par} \vdash: \text{F} \vdash: \text{statement} \\
\text{STT-MODI}
\]

The formal rule of function calls is defined as the rule \text{STT-FUNCALL} below.

\[
\Sigma, \Theta, \Lambda \vdash: \text{type } \Lambda \vdash: \text{option } \text{address } \alpha \vdash: \text{option access } \text{of:option flag } \text{option payable} \\
\Sigma, \Theta, \Lambda \vdash: \text{fpar:call:par} \vdash: \text{e in fpar:call:par} \vdash: \text{STT-FUNCALL}
\]

Here, the result can be either assigned to an \( l \)-value term by defining syntactic sugar or discarded. The variable \text{fpar} is similar to \text{pars} in terms of function declarations rules, and the allowable expressions stored in the list are not limited by type annotation, except that they cannot be an \text{Emodifier} expression. It is noted here that the \text{new} statement in Solidity is treated as a special function call in Lolisa, which can only invoke the constructor function of a contract. In addition, \text{new} can be encapsulated as syntactic sugar.

Loops in Lolisa include the while loop (\text{Loop} \text{while}) and the for loop (\text{Loop} \text{for}), which have been commonly defined in other similar studies [24], and the formal definitions in Lolisa are equivalent. Here, we have not defined the do-while loop. If needed, the do-while loop can be
defined as syntactic sugar using a while loop. The formal rules of Loop\textsubscript{for} and Loop\textsubscript{while} are respectively defined as STT-FOR-LOOP and STT-WHILE-LOOP below.

\[
\Sigma, \Delta \vdash \text{type} \quad F \vdash s, s_1, s_2 : \text{statement} \quad F \vdash e : \text{expr} \land \text{Tbool}
\]

\[
F \vdash \text{Loop}_\text{for} (X : \text{type}, \text{statement} : \text{statement} : \text{statement} : \text{statement}) : \text{statement}
\]

\[
F \vdash e : \text{expr} \land \text{Tbool}
\]

\[
\Sigma, \Delta \vdash \text{type} \quad F \vdash s : \text{statement} \quad F \vdash e : \text{expr} \land \text{Tbool}
\]

\[
F \vdash \text{Loop}_\text{while}(e : \text{expr}) : \text{statement}
\]

\[
F \vdash \text{STT}\text{-FOR-LOOP}
\]

Contract declarations are the most important feature of Solidity, which employs them to declare a class domain. The formal rule of contract declarations in Lolisa is defined as the rule STT-CON below.

\[
F \vdash \text{statement} \quad \Delta \vdash \text{inherits} : \text{list} \quad \Delta \vdash \text{oaddr} : \text{option} \quad \Delta \vdash \text{oa} : \text{option}
\]

\[
F \vdash \text{Contract} : n \text{type} (\text{oaddr} : \text{option} \text{Laddress}), \text{expr} (\text{Tcid} \text{addrr}) : \text{list} \text{Laddress} : \text{statement} : \text{statement}
\]

\[
F \vdash \text{STT}\text{-CON}
\]

Because most higher-order theorem-proving assistants are pure, the inheritance mechanism is formalized based on a module system that is explained in the following section. However, we retain the component of inheritance in the Lolisa syntax for consistency checking, which is also explained in the following section. If the contract declaration is well-formed, its body should not contain any other contract declarations.

\[
\begin{align*}
\text{struct A } \{ & \text{int } a; \quad \text{Struct C } (\text{str}_\text{mem} \text{Tstring } s \text{ str}_\text{nil}) \\
& \text{bool } b; \quad \text{Struct A } (\text{str}_\text{mem} \text{Tint } \text{“} a \text{”}) \\
& \text{struct C } \{ & \text{string } s; \text{ } c; \quad \text{(str}_\text{mem} \text{Tbool } \text{“} b \text{”}) \\
& \}; \quad \text{(str}_\text{mem} (\text{Tstruct C } \text{“} c \text{”} \text{ str}_\text{nil})))
\end{align*}
\]

Figure 7. Equivalent struct declarations in Solidity (left) and Lolisa (right)

Structure declarations are defined as statements because general-purpose high level languages employ structure declarations as a way of defining new composite data types (or record types) based on the requirements of users. Statements str\text{mem} represents a list wrapper of names (defined by string type) and types. In this way, Lolisa provides sufficient flexibility for programmers to define complex customized data types in the formal system. Figure 7 provides an example definition, where the left-hand side is a structure declaration statement in Solidity and the right-hand side is an equivalent declaration in Lolisa, which defines a complex recursive struct type: struct A \{ struct B \{ \ldots ; struct N; \ldots \} \} that effectively demonstrates the facility of the process. The formal type rule for structure declarations is given below by the rule STT-STR.

\[
\Delta \vdash \text{name} : \text{Laddress} \quad F \vdash \text{mem} : \text{extractmem}
\]

\[
\Sigma(\text{mem}) = \emptyset \land \Sigma(\text{mem}) \text{ contains no duplicates}
\]

\[
\forall t \in \Sigma^2(\text{mem}) \exists \Sigma_0, \Sigma_1(\text{mem}) \vdash w(f) \land t \neq \text{Tundef}
\]

\[
F \vdash \text{Struct : Laddress} : \text{extractmem} : \text{statement}
\]

\[
F \vdash \text{STT}\text{-STR}
\]

A struct declaration is well-formed if all of its member types are well-formed and non-void, and if all of the member names are distinct. Finally, struct types with no members are forbidden. We assign \( \Sigma(s) \) to denote a sequence of member names and types associated with a struct tag \( s \). In addition, we separately assign \( \Sigma^1(s) \) to denote the sequence of member names, and \( \Sigma^2(s) \) to denote the sequence of member types associated with struct tag \( s \).

In the following discussion, our meta-variables \( s \), \( s_1 \), \( s_2 \), etc. will range only over well-formed statements satisfying \( w(f) \).

4 Module Semantics

Two problems must be addressed before we can formally define the dynamic semantics of Lolisa syntax. Firstly, Lolisa was developed as a formal subset of Solidity to facilitate contract program execution and verification in higher-order theorem-proving assistants. In contrast to standard program execution, this condition requires that we define all identifiers of the target programs in advance for their use in the source code, and an unambiguous address must consequently be allocated to each identifier. And it is easily to predicate that different functions may have the same variable identifiers pointed to different memory address. Because variables of functions are in different namespace, they are valid to have the same identifiers in real world. However, due to the built-in mechanism, the basic logic context of Coq and similar proof assistants include only a single namespace. Besides, real world programs have been translated into the logic symbols, thus such that identifiers must have unique addresses.
As shown in Table 2, if we want to declare two identical identifiers for two different functions in a single logic context, Coq will return the error message when define the same and it is difficult to formally simulate different namespaces by the type system of Coq. Secondly, because most theorem-proving assistants are pure and based on lambda calculus, expressing the behavior of inheritance or implementing a formal compiler in such theorem-proving assistants is a difficult task. To address these two problems, we introduce the ML module system [27] into Lolisa. This has several advantages. First, a number of higher-order theorem-proving tools, such as Coq, already support the powerful ML module system. Second, the ML module system can define subtyping [28] and subdomains, which can be therefore directly defined in Lolisa. Third, although the ML module system and object-oriented programming languages employ different type systems, they have similar behaviors. For example, sub-modules in Coq can inherit types and members from their super-modules or imported modules. In fact, some mainstream languages, such as Python, support ML module systems and object systems simultaneously. However, the ML module system allows users to access the members of any module if the members are in an equivalent context. Therefore, we retain the component of inheritance in the formal syntax of Lolisa to provide the formal interpreter with inheritance information for checking whether users access invalid modules while writing theorems manually.

Of course, the best means of avoiding this is to associate a single main contract module and its related modules with a single context.

### Syntax

\[
\Gamma[M] :::= L | K
\]

\[
L ::= \text{contract } C_0 \text{ is } C \ (N \{ R \}; \ \bar{R})
\]

\[
N ::= \text{function } F \ [\bar{R}]
\]

\[
K ::= id | p.id | this.id | C
\]

### Module

\[
\mathcal{M} ::= C | F | C(F) | C[\mathcal{M}] | C[\mathcal{M}_0; \ldots; \mathcal{M}_n]
\]

### Subtyping

\[
C <: C \ (\text{BAS-C}) \ F <: F \ (\text{BAS-F})
\]

\[
\frac{C_0 \subseteq C_1 \land C_1 \subseteq C_2 \ (\text{TRANS}) \lor \ i \in \mathbb{N}, \text{contract } C_0 \text{ is } C_1 \ (\ldots) \ G_i \subseteq C_i \ (\text{INHERT})}{contract C \ C (\text{function } F \ldots)}
\]

### Access

| Syntax |
|--------|
| \[
\Gamma[M] + p : \text{Module} \ id_1 \ldots id_n End \\
\Gamma[M] + call(id) = call(p.id)
\] (SING) |
| \[
\Gamma[M] + p : \text{Module} \ id_1 \ldots id_n End \\
\Gamma[M] + call(id) = call(\text{M}_\text{nearest.id})
\] (MULT-IN) |
| \[
\Gamma[M] + p : \text{Module} \ id_1 \ldots id_n End \\
\Gamma[M] + call(id) = call(id)
\] (MULT-OUT) |
| \[
\Gamma[M] + p : \text{Module} \ id_1 \ldots id_n End \\
\Gamma[M] + call(id) = call(\text{M}_\text{nearest.id})
\] (MULT-NOT-IN) |
| \[
\Gamma[M] + p : \text{Module} \ id_1 \ldots id_n End \\
\Gamma[M] + \text{call(id)} = \text{call(\text{M}_\text{nearest.id})}
\] (MULT-OUT) |
| \[
\Gamma[M] + F : \text{C} \subseteq C \subseteq \text{C}_1 \subseteq \ldots \subseteq \text{C}_n \subseteq \text{C} \ (\text{THIS-T})
\] |
| \[
\Gamma[M] + call(id) = \text{call(\text{id})}
\] |
| \[
\Gamma[M] + \text{call(id)} = \text{call(\text{id})}
\] (THIS-F) |

Figure 8. Module declaration abstraction syntax, declarations, and member access type assignment rules of Lolisa

Lolisa has two kinds of modules: contract modules and function modules. The members of contract and function modules are specified by the identifier and Lolisa source code \( C \), and we assign \( \Gamma[M] \) to denote the current global context. The abstract syntax of Lolisa in terms of \( \Gamma[M] \), contract module declarations \( L \), function module declarations \( N \), and member declarations \( K \) is given at the top of Figure 8. The meta-variables \( C, C_0, C_1, C_2, \ldots \) range over contract module names; \( F, F_0, F_1, F_2, \ldots \) range over function module names; \( M \) and \( p \) range over arbitrary module sets and terms with module types; \( id \) ranges over identifiers. We assign \( \mathcal{M} \) to denote possible sequences \( \mathcal{M}_0, \ldots, \mathcal{M}_n \), and assign \( \bar{C}, F, \) and \( \bar{id} \) to denote similar possible sequences for \( C, F, \) and \( id \), respectively. The predicate \( call(id) \) denotes the act of the source code accessing (applying) the respective identifier. According to the abstract syntax, it is clear that a function module must be in a contract module. In other words, a function module is forbidden in Lolisa to be defined independently, and a contract module is forbidden to be defined in a function module.
The module semantics can now be defined based on the abstract syntax shown in Figure 8. Modules are constructed by contracts, functions, and their recursive combinations. A subtype relationship between modules is denoted by $A < : B$, where $A$ is a subtype of $B$. Therefore, the rules BAS-C and BAS-F indicate that contract modules and function modules basically represent reflexive closure, while the rule TRANS indicates that contract modules also represent transitive closure. In addition, the rules INHERT and IN-FUN indicate that a main contract module is the subtype of an inherited module (in Solidity, the keyword is denotes inheritance) and a function module is the subtype of its super-contract module. The remaining rules represent member access type assignments. The rule SING defines that the operation call(id) for an arbitrary identifier id in a module $p$ is equivalent with accessing the id of $p$, i.e., call(p.id). This follows equally if other super-modules $\tilde{M}$ and $p$ have an equivalent identifier id. Therefore, we do not check whether other super-modules have equivalent identifiers, as given by the rule MULT-IN.

When the identifier id exists in $\tilde{M}$ but not exist in $p$, the source code will access the identifier in the nearest module, as given by the rule MULT-NOT-IN. Alternatively, as given by the rule MULT-OUT, the source code can access an id in $\tilde{M}$ directly, denoted as call($\tilde{M}$.id). The special identifier this employed in Solidity and other object-oriented languages is accommodated in Lolisa according to the rules THIS-T and THIS-F. Here, this serves a single function, in that, if a function module $F$ has an identifier combined with this (i.e., this.id), the identifier in the super-contract module of $F$ is accessed (i.e., call(C.id)). Otherwise, an incorrect result will be obtained.

We have formalized the domain separation and inheritance behavior of Lolisa. The following discussion of dynamic semantics are all based on these developed module semantics.

5. Formal Semantics

The deep correspondences applied in CHI make it very useful for unifying formal proofs and program computation. In brief, CHI proposes that a deep correspondence exists between the world of logic and the world of computation. This correspondence can be expressed according to three general principles: propositions as types, proofs as programs, and proofs as evaluation of programs. However, as discussed, most higher-order logic theorem-proving assistants are based on lambda calculus; yet, most mainstream programming languages employed in the real world are not designed based on lambda calculus, and cannot be analyzed in a higher-order logic environment. The programs written using these languages are very difficult or even impossible to verify directly and automatically using CHI. This forms the basis of EVI [3], which extends the formal relations of CHI to include three corollaries: proofs as evaluation of programs as execution of programs, properties as propositions as types, and verifications as proofs. Based on these corollaries, the correspondences of CHI can be made still deeper to obtain a fourth general principle: verifications as execution of programs. Accordingly, Lolisa is defined to be a formal subset of Solidity that can be executed and reasoned in higher-order logic theorem-proving assistants based on EVI. Thus, the formal semantics are based on the GERM framework, in conjunction with memory management operations conducted with APIs defined in the GERM framework. Therefore, we can guarantee a safe-type memory access because every memory block of the GERM framework stores the native logic information directly, and employs specific types for different memory values.

We now formalize the dynamic semantics of Lolisa using natural semantics, which are also known as big-step operational semantics [4]. Because the static formal semantics of Lolisa (that is, its formal syntax typing rules) have been formally specified, the dynamic semantics are defined under the assumption that programs written in Lolisa are well-typed, and, in particular, with the assumption that the type annotations of expressions are consistent. However, programs written in Lolisa also include undefined behaviors when accessing memory space owing to execution authority or other limitations. Clearly, we must know whether the behavior of programs is correct if we wish to reason in a higher-order logic world. Therefore, with reference to the basic API definitions of the GERM framework, we employ a monad-type option [30] to represent the different conditions represented by return values. Here, the return value is annotated as Some if it is meaningful, None if it is nothing, and, otherwise, it is assigned an error message Error.

5.1 Predefinitions of semantics evaluations

Each subsection of the semantics defines its own operators and miscellaneous notation. However, a number of general observations can be given here regarding notation. All the following subsections present an evaluation relation of the form $\sigma_0 \Downarrow v_0 (\nu, \sigma_1)$, where $\sigma_0$ and $\sigma_1$ are the initial and final memory states, respectively, $v_0$ represents the form of Lolisa syntax being defined, and the nature of $v_1$ depends on the precise evaluation relation being defined. We employ the notation $\alpha \sqsubseteq \{b\}$ to indicate that the term $\alpha$ will be at least simplified as a kind of normal form existing in the set $b$. 
map_addr | Searches the indexed address of a mapping type | map_get | Obtains the value stored in a mapping type term
--- | --- | --- | ---
evallop | Evaluates binary operation expressions | eval_wop | Evaluates unary operation expressions
memfind | Searches the required struct member | env_check | Validates the current environment
setenv | Changes the current environment | inherit_check | Validates the inheritance information
initvar | Initializes the variable address | init_re | Initializes the function return address
istrue | Predicate for a true value in Lolisa | isfalse | Predicate for a false value in Lolisa

| Statement outcomes: out := | | | |
| stop | stop executing current statement | stop | stop executing current statement with error message |
| error | function exit | exit | function exit, returning the value v |
| exit(v) | go to the next iteration of the current loop | continue | go to the next iteration of the current loop |
| break | exit from the current loop | break | exit from the current loop |

Optional return value: opm ::= Some v | return a meaningful value v |
None | return nothing |
Error | return an error message |

Table 3. Helper functions

Table 3 respectively summarize the helper functions used in the dynamic semantic definitions. One of the important aspects of the dynamic semantics of Lolisa is the environment information used to observe changes in the environment, and to determine whether programs are executing in valid environments. This information is composed of two environment information terms: the current environment env, which stores the current execution environment information, and the super-environment fenv, which stores the super environment information of env. In the initial state, env and fenv are equivalent, except for the gas value because env stores the remaining amount of gas and fenv stores the minimal gas limit. Then, the helper functions listed in Table 3 are usually used as abbreviations for relatively complicated expressions regarding states, but are not particularly interesting in their own right. A few of these functions and their components will be defined in the course of this section.

5.2 Evaluation of Values

The formal semantics of value evaluations involve evaluating Lolisa values, obtaining native value information that can be computed or reasoned in the base formal system, and generating respective memory values in the GERM framework. Here, we adopt the meta-symbol val to represent both a Lolisa value val and a mapping value val_map. This is possible because val_map has equivalent static typing rules as val, except for not having definitions related to mapping types. In addition, each value maps to a unique respective memory value, as discussed in Section 2, and we assign the symbol \( \equiv \) to indicate the map relation. No value evaluations change the memory state \( \sigma \), and we specify the evaluation of a value under \( \sigma \) as \( \sigma \triangleright val \).

We first define the evaluation semantics of normal form values. As discussed, normal form values refer to Lolisa values that can access the respective native value information of the formal system directly. Therefore, based on the design of the GERM framework, normal form values can be evaluated to obtain the respective memory values directly, as given by the rule EVAL-V-CONS below.

\[
\begin{align*}
E & \triangleright env.fenv \quad M \mapsto \sigma.binf
\\
\Sigma, \lambda, b_{inf}, \forall n: Type.function(var(n)) \land \forall n: evalf
\\
val(n) & \equiv m_{value}
\\
E, M, \Sigma, \lambda, b_{inf} & \triangleright (\sigma, env.fenv, val(n)) \equiv (\sigma, env.fenv, m_{value}(n, env.b_{inf})))
\end{align*}
\]

(EVAL-V-CONS)

Secondly, we define the evaluation semantics of non-normal form values, which are employed to access the memory space, match the key or index, and obtain the expected values, as given below by the rules EVAL-V-ARR1, EVAL-V-ARR2, and EVAL-V-ARR3, respectively, which illustrate the natural semantics of array values.

\[
\begin{align*}
E & \triangleright env.fenv \quad M \mapsto \sigma.binf
\\
F, b_{inf} & \triangleright \text{Array}(\lfloor d_{array} \rfloor) \quad A \triangleright \text{name}
\\
\sigma \triangleright id_{array}(\text{offset}, \sigma) & \rightarrow \text{address} + \text{offset} = \text{name} \rightarrow \text{Error}
\\
E, M, F & \triangleright (\sigma, env.fenv, \text{Array}(\text{name} \lfloor d_{array} \rfloor) \equiv (\sigma, env.fenv, \text{Error})
\end{align*}
\]

(EVAL-V-ARR1)
can be value and obtain the stored value through parameters. Second, error message.

The field access address to which it pertains needs to be extracted by a rule EXP-ARR-N'. We should check the type to determine whether information exists for the next dimension. According to verification of the basic APIs of the GERM framework [3], we can assume that, if read_check is invoked successfully, the type and the memory value satisfy the map relation. Then, we should call the rule EVAL-V-ARR2 again like the rule EVAL-V-ARR3.

For mapping values, the evaluation process has three parts. First, the operation \( \psi_{id_{array}} \) yields the key value \((id: in\text{val } \tau_{map})\), and read_check attempts to extract the mapping value stored in the initial address. If \( \psi_{id_{map}} \) or read_check fails, the evaluation process will return an error message. Otherwise, the evaluation process will be simplified as \( map_{addr} \), and it will take the results of \( \psi_{id_{map}} \) and read_check as parameters. Second, if \( map_{addr} \) successfully obtains the memory block whose key is equal to \( id_{map} \), then read_dir will extract the mapping value and obtain the stored value through \( map_{get} \). Here, we have ensured by read_check that the memory block stores a mapping value, so it can be extracted by read_dir directly. Finally, if a next dimension exists and the result of \( map_{get} \) is Map, then the next dimension is evaluated; otherwise, an error message is returned. The evaluation semantics of mapping values are defined according to the following rules.

At the value level, struct is only employed to represent a memory value with a struct type. Therefore, it is similar to a normal form value, and we need only extract the struct value by read_check directly, as given by the rule EVAL-V-STR below.
functions. For example, a struct field access $A.B.f(a,b,c).C$, where $f(\cdot)$ is a function, is forbidden in Lolisa. Here, we need not worry about the built-in EVM functions in a standard structure, such as $msg$ or $block$, because these have been defined in the Lolisa standard library in advance, as will be discussed later; therefore, they can be treated as normal structures in the semantics. We assign $\mathcal{u}_{\text{field}}$ to denote the process of evaluating a base address $a_{\text{init}}$ and a struct-type address $a_{\text{type}}$. Then, $\text{mem}_{\text{find}}$ will take $\text{mem}_0, \text{mems}_m, a_{\text{init}},$ and $a_{\text{type}}$ as parameters and attempt to obtain the memory value indexed by $\text{mem}_n$. If $\text{mem}_{\text{find}}$ is invoked successfully, it returns a pair $(\text{Dad}, m_v)$. Here, $\text{Dad}$ actually refers to the address of $\text{mem}_{n-1}$ because Solidity will take $\text{mem}_{n-1}$ as an implicit argument if $\text{mem}_n$ is a function call pointer, such as $a.\text{send}(v,mss) \equiv \text{send}(a,v,mss)$.

This indicates that the identifier $a$ is a parameter of the $\text{send}$ function during interpretation or compilation, which is a common usage in Solidity. Therefore, if $m_v$ is a function pointer $(\text{Fid}(\ast))$ for a function call, $\text{Dad}$ and function input $\text{opars}$ should be combined into $m_v$ to facilitate their transmission to the next level. And the above evaluation process are summarized as following semantics $\text{EVAL-V-FIELD-T}$ and $\text{EVAL-V-FIELD-F}$.

Finally, Solidity includes some special cases where some built-in EVM functions employ some special variable types, such as a mapping type variable $v$ to invoke the built-in function $\text{length}(v.\text{length})$. This condition is also addressed by Lolisa standard library functions, which is discussed in the Subsection 5.5, below.

5.3 Evaluation of Expressions

The semantics of expression evaluation are the rules governing the evaluation of Lolisa expressions into the memory address values of the GERM framework, and this process includes two parts: the l-value position evaluation and the r-value position evaluation. In contrast, modifier expressions are a special case that cannot be evaluated according to these expression evaluation semantics, but their evaluation is conducted according to the following semantics statement.

\begin{equation}
\mathcal{u}_e \langle \sigma, \text{env}, \text{fenv}, \text{Emodifier}(\ast) \rangle \Rightarrow \langle \sigma, \text{env}, \text{fenv}, \text{Error} \rangle \quad (\text{EVAL-EXP-MODI})
\end{equation}

Here, $\mathcal{u}_e$ represents the process of evaluating a modifier expression both in l-value position and r-value position.

Evaluating expressions in the l-value position In the following, we assign $\mathcal{u}_e^l$ to denote the evaluation of expressions in the l-value position to yield respective memory addresses. First, obviously, most expressions constructed by $\text{Econst}$ constructor cannot be employed as the l-value. Because most of them represent a Lolisa constant value at expression level directly. For example,

$$\text{Econst}(V\text{bool}(\text{true})): \text{exprbool}V\text{bool}$$

represents a Boolean value at expression level. However, specially, in Lolisa, the l-value positions are allowed to employ the constant expressions specified by value constructors $V\text{array}$ and $V\text{map}$. As mentioned previously, value constructors $V\text{array}$ and $V\text{map}$ are actually address pointers for indexing specific memory blocks. For example, $A[i] = a$ is a common use in most general-purpose programming languages. Thus, they not only can represent memory values at the value level, but can also represent memory addresses at the expression level. This can facilitate a simplification of the Lolisa syntax for brevity, we assign $A[a, \ast]$ to denote the recursive processes of $V\text{array}$ and $V\text{map}$ employed for searching indexed addresses, as defined in the previous subsection. Note that $V\text{struct}$ and $V\text{field}$ are forbidden to specify expressions in the l-value position, even though they are also address pointers. This is because, for $V\text{struct}$, $V\text{var}$ can represent any variable address for any type. Therefore, to avoid confusion, $V\text{struct}$ can only represent a memory value at the value level, as represented by Convention 1. For $V\text{field}$, both Solidity and Lolisa include a number of special structures, such as $msg$ and $block$, and their members are forbidden to be altered arbitrarily. Moreover, in most cases, it is invalid in Solidity to modify the values stored in a field. Therefore, to ensure that Lolisa is well-formed and well-behaved, $V\text{field}$ is forbidden from specifying expressions in the l-value position. The only means allowed in Lolisa of altering the fields of structures are using $E\text{struct}$ to change either all fields or declaring a new field. Although this limitation may be not friendly for programmers or verifiers, it avoids potential risks.

In the previous subsection, we defined the semantics of array values. Accordingly, we can define the address searching process based on the
semantics of arrays as the rule ARR-SEARCH below, which takes name, \( \mathbf{a}_{i_darray} \), \( \mathbf{t} \), and \( address\_offset \) as parameters. Similarly, we can define the rule MAP-SEARCH below for mapping values.

\[
\mathcal{A}[\sigma, name, \mathbf{a}_{i_darray}, \mathbf{t}, address\_offset] \quad \text{(ARR-SEARCH)}
\]

\[
\mathcal{A}[\sigma, name, \mathbf{a}_{i_map}, \mathbf{t}, map\_sn] \quad \text{(MAP-SEARCH)}
\]

Taking array values as an example, as given by the rules EVAL-LEXP-ARR1 and EVAL-LEXP-ARR2 below, if ARR-SEARCH obtains the address successfully, the address can be transmitted to the statement level; otherwise, an error message is returned. We similarly define the rules of mapping values, as given by the rules EVAL-LEXP-MAP1 and EVAL-LEXP-MAP2 below. The rule EVAL-LEXP-OTHER-CONS below indicates that constant expressions specified by the Lolisa value constructors other than Varray and Vmap are invalid. The expressions Extract, Eref, and Eup which are summarized as eaddr in Figure 5, need only return their address directly, as given by the rule EVAL-LEXP-ADDR below.

Evaluating expressions in the r-value position In the following, we assign \( \mathcal{S}_r \) to denote the evaluation of expressions in the r-value position to yield respective memory addresses. The rules EVAL-REXP-CONS1 and EVAL-REXP-CONS2 below define the evaluation of constant expressions.

\[
\mathcal{M} \vdash (\sigma, env, fenv)(\mathcal{S}_r(\mathcal{S}_l(e_{value}))) \quad \text{(EVAL-REXP-CONS1)}
\]

\[
\mathcal{M} \vdash (\sigma, env, fenv)(\mathcal{S}_r(\mathcal{S}_l(e_{value}))) \quad \text{(EVAL-REXP-CONS2)}
\]

Here, we note that, because constant expressions store Lolisa values directly, the results can be obtained by applying \( \mathcal{S}_r \) directly. In the expression level, the r-value position is specified with a struct type. This is also the only means of initializing or changing the value of a struct-type term. The rules EVAL-REXP-STR1 and EVAL-REXP-STR2 below define this process.

\[
\mathcal{E} \vdash \mathcal{S}_r(e_{struct}) \quad \mathcal{M} \vdash \mathcal{S}_r(e_{struct}) \quad \text{(EVAL-REXP-STR1)}
\]

\[
\mathcal{E} \vdash \mathcal{S}_r(e_{struct}) \quad \mathcal{M} \vdash \mathcal{S}_r(e_{struct}) \quad \text{(EVAL-REXP-STR2)}
\]

Here, if the evaluation of Extract fails, the process of evaluating a member's value yields an error message. Otherwise, the member's value set is obtained and the respective struct memory value is returned. The semantics of address pointer expressions, the results can be obtained by applying \( \mathcal{S}_r \) directly as
EVAL-REXP-ADDR1. In particular, the expression $Efun$ should read the function return address $A_{\text{fun}}$ to obtain the result of the respective function pointer as EVAL-REXP-ADDR2. Finally, the semantics of binary and unary operations are defined according to the rules EVAL-REXP-BOP, EVAL-REXP-UOP, and EVAL-REXP-OP-F below.

\[
\frac{\mathcal{E}, M \vdash s \in \text{addr}}{\mathcal{E}, M \vdash \text{addr}(\alpha, s) \land \mathcal{E}(\text{addr}(\alpha, s))}{\text{eval}} \quad \text{(EVAL-REXP-BOP)}
\]

\[
\frac{\mathcal{E}, M \vdash \alpha \text{env} \land \mathcal{E}(\alpha \text{env})}{\mathcal{E}, M \vdash \text{envexp}(\alpha \text{env}) \land \mathcal{E}(\text{envexp}(\alpha \text{env}))}{\text{evalu}} \quad \text{(EVAL-REXP-UOP)}
\]

\[
\frac{\mathcal{E}, M \vdash \text{evalu}(\alpha \text{env}, m_0) \land \mathcal{E}(\text{evalu}(\alpha \text{env}, m_0))}{\text{evalu}(\alpha \text{env}, m_0) \text{ re}} \quad \text{(EVAL-REXP-OP-F)}
\]

Due to the static type limitations in the formal abstract syntax definition based on GADTs, the expressions, sub-expressions, and operations are all guaranteed to be well-formed, and the type dependence relations need not be checked using, e.g., informal assistant functions, as required by other formal semantics such as Clight. The functions $\text{evalu}_{\text{bop}}$ and $\text{evalu}_{\text{uop}}$ take the results of expression evaluations and required operations as arguments, and combine them together to generate new memory values. The example given in Table 4 is the cases corresponding to binary addition.

| $v_0$ | $v_1$ | $\text{evalu}_{\text{bop}}(M, +, v_1, v_2)$ |
|-------|-------|----------------------------------|
| $\text{init}(v_0)$ | $\text{int}(v_1)$ | $\text{Some } \lfloor v_0 + v_1 \rfloor$ |
| $\text{float}(v_0)$ | $\text{float}(v_1)$ | $\text{Some } \lfloor v_0 + v_1 \rfloor$ |
| otherwise | Error |

Table 4. Case study

In the present version of Lolisa, the above definition forbids mixed arithmetic operations, such as “int + float”, because, as discussed previously. Solidity does not completely support the float datatype, and float values are also rarely employed in smart contract programs. Therefore, including mixed arithmetic operations will add unnecessary complexity and computational burden to formal interpreter or compiler implementation. Of course, the semantics do not forbid the use of mixed arithmetic operations, and the formal syntax of expressions also preserves sufficient extensibility, such that users can extend Lolisa themselves accordingly if deemed necessary. In addition, the results of $\text{evalu}_{\text{uop}}$ and $\text{evalu}_{\text{bop}}$ should be signed as a monad-type option.

Although the formal syntax of expressions is designed using GADTs, and the results can be assumed to be well-formed and follow the type rules statically without invalid combinations, invalid behaviors can occur when evaluating values. For example, the simplest and most obviously invalid case is a divisor of 0 in division operations. The specific design target of Lolisa indicates that we should formally address division by zero. As such, we adopt a return value of $\text{Some}$ to express valid division results, and $\text{Error}$ to represent undefined results caused by invalid division, which is expressed by rules EXM-1 and EXM-2 below.

\[
\frac{\alpha \vdash v_0 \_v_1}{\_v_0 = 0} \quad \text{(EXM-1)}
\]

\[
\frac{\alpha \vdash \_v_0 \_v_1}{\_v_0 = 0} \quad \text{(EXM-2)}
\]

The evaluation processes of other operations are similarly conducted.

5.4 Evaluation of Statements

In the following, we assign $\psi_{\text{stt}}$ to denote the evaluation process of statements. Most evaluations employ two helper functions $\text{envcheck}$ and $\text{setgas}$. The helper function $\text{envcheck}$ takes the current environment $\text{env}$ and the super-environment $\text{fenv}$ as arguments, to check conditions such as gas limitations and the congruence of execution levels. For example, if the domains in $\text{env}$ and $\text{fenv}$ are equal, but the execution levels differ, the program will be stopped, and $\text{env}$ will be reset by $\text{fenv}$. This is formally defined in the rule ENV-F below, where current statements will be executed only if the result of $\text{envcheck}$ is true; otherwise, the program will stop and return to the beginning memory state.

\[
\frac{\mathcal{E}, M \vdash \_v_0}{\mathcal{E}, M \vdash \text{envcheck}(\_v_0) = \text{false}} \quad \text{(ENV-F)}
\]

The helper function $\text{setgas}$ takes current execution statement and environment as arguments to deduct the gas recorded in $\text{env}$ and generates a new environment $\text{env}'$ with a new gas amount. In Coq, $\text{setgas}$ is actually implemented as a matching tree whose branches are the deduction cases of gas following the gas price sheet in [40]. And, defined as the rule GAS-F, if the $\text{setgas}$ fails, the current program execution will be stopped.

\[
\frac{\mathcal{E}, M \vdash \_v_0}{\mathcal{E}, M \vdash \text{gas}(\_v_0) = \text{None}} \quad \text{(GAS-F)}
\]

Contract declarations are one of the most important statements of Solidity. In Lolisa, contract declaration involves two operations. First, the consistency of inheritance information is checked using the helper function $\text{inheritcheck}$, which takes the inheritance relations in module
Variable declarations are one of the most basic statements in Lolisa. As discussed in Section 4, the namespace of identifiers is controlled by the ML module system and stored in env and fenv. Therefore, with variable declarations, we can use env and fenv directly. The function \( \text{init}_{\text{par}} \), which is a special case of \( \text{write}_{\text{dir}} \), takes the variable type, indexed address, and environment information as parameters, and initializes respective memory blocks, which is defined as the rule EVAL-STT-VAR below.

\[
\begin{align*}
\text{Env} & \vdash \text{env} \text{.fenv} & \text{M} = \sigma \cdot \text{binfor} \quad \text{F} \cdot \text{binfor} \vdash v \\
\Sigma, \text{A} & \vdash \text{binfor} \vdash \text{Evar}(\text{name}) \\
\text{EM} & \vdash \text{envcheck} (\text{env} \cdot \text{fenv}) & \text{true} \land \text{setenv}(\text{env} \cdot \text{fenv}) \Rightarrow \text{some env}' \\
\text{write}_{\text{dir}} \left( \sigma, \text{str}, \text{strtype}(\text{str}, \text{mem}, \text{env} \cdot \text{binfor}) \right) & \Rightarrow \sigma' \\
\text{EM} & \vdash (\text{env} \cdot \text{fenv} \cdot \text{Struct}(\text{str}, \text{mem}, \text{Evar}(\text{name}))) \Rightarrow (\sigma' \cdot \text{env} \cdot \text{fenv} \cdot \text{stop})
\end{align*}
\]  

(EVAL-STT-VAR)

Here, we assume that the current logical context based on the GERM framework includes sufficient logical memory space, such that each identifier has a valid and free address. Therefore, all indexed memory blocks have been initialized by \( \text{init}_{\text{par}} \), and \( \text{write}_{\text{dir}} \) will be always successful.

With respect to struct declarations, a new struct type is declared in the statement level according to Convention 1. The address is the new struct type identifier, and the struct type information is written into the respective memory block directly. The dynamic semantics of this process are defined by the rule EVAL-STT-STRUCT below.

\[
\begin{align*}
\text{Env} & \vdash \text{env} \text{.fenv} & \text{M} = \sigma \cdot \text{binfor} \\
\Sigma, \text{A} & \vdash \text{binfor} \vdash \text{str} \\
\text{EM} & \vdash \text{envcheck} (\text{env} \cdot \text{fenv}) & \text{true} \land \text{setenv}(\text{env} \cdot \text{fenv}) \Rightarrow \text{some env}' \\
\text{write}_{\text{dir}} \left( \sigma, \text{str}, \text{strtype}(\text{str}, \text{mem}, \text{env} \cdot \text{binfor}) \right) & \Rightarrow \sigma' \\
\text{EM} & \vdash (\text{env} \cdot \text{fenv} \cdot \text{Struct}(\text{str}, \text{mem}, \text{Evar}(\text{name}))) \Rightarrow (\sigma' \cdot \text{env} \cdot \text{fenv} \cdot \text{stop})
\end{align*}
\]  

(EVAL-STT-STRUCT)

In Lolisa, a function call statement is used to define the function body indexed by the call statement. The process of applying an indexed function is defined by the rules EVAL-STT-FUN-CALL-T and EVAL-STT-FUN-CALL-F below.

\[
\begin{align*}
\text{Env} & \vdash \text{env} \text{.fenv} & \text{M} = \sigma \cdot \text{binfor} \\
\Sigma, \text{A} & \vdash \text{binfor} \vdash \text{Env} (\text{some a}) \\
\text{EM} & \vdash \text{envcheck} (\text{env} \cdot \text{fenv}) & \text{true} \land \text{setenv}(\text{env} \cdot \text{fenv}) \Rightarrow \text{some env}' \\
\text{read}_{\text{dir}} \left( \sigma, \text{env} \cdot \text{binfor} \cdot \text{a} \right) & \Rightarrow \text{some m}_\text{a} \\
\text{setenv}(\text{env} \cdot \text{a} \cdot \text{b}) & \Rightarrow \text{env}' \\
\text{EM} & \vdash (\text{env} \cdot \text{fenv} \cdot \text{Fun-call} (\text{Efun}(\text{some a}) \cdot \text{inputs})) \Rightarrow (\sigma, \text{env} \cdot \text{fenv} \cdot \text{stop})
\end{align*}
\]  

(EVAL-STT-FUN-CALL-T)

\[
\begin{align*}
\text{Env} & \vdash \text{env} \text{.fenv} & \text{M} = \sigma \cdot \text{binfor} \\
\Sigma, \text{A} & \vdash \text{binfor} \vdash \text{Env} (\text{some a}) \\
\text{EM} & \vdash \text{envcheck} (\text{env} \cdot \text{fenv}) & \text{true} \land \text{setenv}(\text{env} \cdot \text{fenv}) \Rightarrow \text{some env}' \\
\text{read}_{\text{dir}} \left( \sigma, \text{env} \cdot \text{binfor} \cdot \text{a} \right) & \Rightarrow \text{Error} \\
\text{EM} & \vdash (\text{env} \cdot \text{fenv} \cdot \text{Fun-call} (\text{Efun}(\text{some a}) \cdot \text{inputs})) \Rightarrow (\sigma, \text{env} \cdot \text{fenv} \cdot \text{stop})
\end{align*}
\]  

(EVAL-STT-FUN-CALL-F)

Here, the first step is an attempt to extract the function declaration statements stored in the respective memory address by \( \text{read}_{\text{dir}} \). If the read operation is successful, the second step sets the current execution environment level as 0, and also sets the domain as the called function identifier using \( \text{setenv} \). The final step executes the function body by \( \sigma \cdot \text{m}_{\text{env}}(\text{a}) \) with the new environment \( \text{env}' \).

Modifier declarations are a kind of special function declaration that require three steps, and includes a single limitation. The parameter values are set by the \( \text{setpar} \) predicate. As defined by the rule EVAL-STT-MODI below, the first step (denoted as \( \oplus \)) initializes and sets the parameters. The second step (denoted as \( \otimes \)) stores the modifier body into the respective memory block. The third step (denoted as \( \odot \)) attempts to initialize the return address \( \text{A}_{\text{fun}} \). Due to the multiple return values, \( \text{init}_{\text{re}} \) takes a return type list as an argument. Then, we attempt to execute the modifier body. Here, although not formally expressed in the rules below, an attempt to execute the body returns an error if \( \oplus = \text{Error} \lor \otimes = \text{Error} \lor \odot = \text{Error} \). According to the rule MODI-LIMIT below, the modifier body can only yield an initial memory state, and therefore cannot change memory states. Due to the separate definitions given by the rules STT-FUN and STT-FUNS discussed in Subsection 3.4, the dynamic semantics of function declarations are also defined separately below by the rules EVAL-STT-FUN-T, EVAL-STT-FUN-F, EVAL-STT-FUNS-T, and EVAL-STT-FUNS-F. The difference between modifier semantics and function semantics is modifier limitations checking. Specifically, before invoking a function, the modifier, restricting the function, will be executed, and the results of modifier evaluations will be checked. If the result is \( \sigma_{\text{init}} \), it means the limitations checking of the modifier fails and the function invocation will be thrown out. Otherwise, the function will be executed.
The semantics of conditional statements in Lolita are similar to those of other well-known works, where, in the following rules EVAL-STTIF1, EVAL-STTIF2, and EVAL-STTIF3, $\mathbf{M}_e$ denotes the evaluation process yielding a Boolean result.

\[
\begin{align*}
\text{EVAL-STTIF1} & : \quad \mathbf{M}_e \vdash e \quad \text{EVAL-STTIF2} : \quad \mathbf{M}_e \vdash e \quad \text{EVAL-STTIF3} : \quad \mathbf{M}_e \vdash e
\end{align*}
\]

The semantics of assignment statements in Lolita are similar to those of other well-known works. In particular, if the result of an $r$-value

\[
\begin{align*}
\text{EVAL-STTSEQ1} & : \quad \mathbf{M}_e \vdash s_1 \quad \text{EVAL-STTSEQ2} : \quad \mathbf{M}_e \vdash s_1
\end{align*}
\]
expression evaluation is a kind of function pointer generated by field access, then the function call semantics $EVA\text{L}$-STT-FUN/S-T/F defined above are applied to evaluate the return values as $EVA\text{L}$-STT-ASSIN1. Otherwise, as illustrated in $EVA\text{L}$-STT-ASSIN2 and $EVA\text{L}$-STT-ASSIN3, the evaluation process of assignment statements will try to evaluate the memory address $\alpha$ of l-value expression by $\Psi_\ell^\alpha$, and the memory value $v$ of r-value expression by $\Psi_\ell^v$. If the evaluation of l-value and r-value both are successful, then $v$ will be written in $\alpha$ through employing $\text{write}_\text{check}$. The semantics of assignment statements are defined according to the following rules.

$$EVA\text{L}$-STT-ASSIN1$$

$$EVA\text{L}$-STT-ASSIN2$$

$$EVA\text{L}$-STT-ASSIN3$$

The semantics of for and while loops in Lolisa are also similar to those of other well-known works, where the semantics incorporate four conditions: 1) continuing to the next loop; 2) exiting from the current loop due to loop failure; 3) breaking from the current loop; 4) exiting from the current loop due to an error message. These four conditions are appended to the corresponding rules $EVA\text{L}$-STT-FOR and $EVA\text{L}$-STT-WHILE below.

$$EVA\text{L}$-STT-FOR1$$

$$EVA\text{L}$-STT-FOR2$$

$$EVA\text{L}$-STT-FOR3$$

$$EVA\text{L}$-STT-WHILE1$$

$$EVA\text{L}$-STT-WHILE2$$

$$EVA\text{L}$-STT-WHILE3$$

$$EVA\text{L}$-STT-WHILE4$$

Here, we need only concentrate on single statement evaluation rather than distinguishing between infinite and finite loops. Cases involving finite and infinite loops are differentiated as part of the macroscopic process of program evaluation, which will be discussed in the following subsection.

The semantics of return and multi-value return expressions are evaluated according to the semantics of expressions, as respectively defined
by the rules EVAL-STT-RE and EVAL-STT-RES below. First of all, the Return semantics will try to evaluate the result of returning expression \( e \) with \( \mathcal{U}_e \). And the semantics of statements Returns are similar to the semantics of statements Return that try to evaluate all of the expressions recorded in heterogeneous list \( e_s \) by employing \( \mathcal{U}_e \) iteratively.

\[
\begin{align*}
\epsilon \cdot \text{env}, \text{fenv} & \leftarrow \sigma, \mathrm{bin}_{\text{for}} \times \mathcal{E}, \mathcal{F}^+ \cdot e \quad \text{env}_c(\text{env}, \text{fenv}) = \text{true} \land \text{set}_\text{zam}(\text{Return}(e), \text{env}) = \text{Some env}' \land \sigma \mathcal{U}_e(\sigma, \text{some v}) \\
& \quad \text{write}_\text{di}(\text{env}, \text{fbin}_{\text{for}}, \text{Afun}, \text{env}) = v' \\
& \quad \text{set}_\text{zam}(\text{env}', \text{fenv}(\text{domain})) = \text{env}' \\
\mathcal{E}, \mathcal{F}^+ = (\sigma, \text{env}, \text{fenv}, \text{Return}(e)) = (\sigma, \text{some env}', \text{fenv}, \text{error}) & \quad \text{(EVAL-STT-RE1)} \\
\epsilon \cdot \text{env}, \text{fenv} & \leftarrow \sigma, \mathrm{bin}_{\text{for}} \times \mathcal{E}, \mathcal{F}^+ \cdot e = \epsilon_1, \epsilon_2, \ldots, \epsilon_n \\
& \quad \text{env}_c(\text{env}, \text{fenv}) = \text{true} \land \text{set}_\text{zam}(\text{Return}(e), \text{env}) = \text{Some env}' \\
& \quad (\text{repeat } \sigma \mathcal{U}_e(\sigma, \text{env})) = \epsilon_1, \epsilon_2, \ldots, \epsilon_n \sim [\mathcal{O}_1, \mathcal{O}_2, \ldots, \mathcal{O}_n] \\
& \quad \text{write}_\text{di}(\text{env}, \text{fbin}_{\text{for}}, \text{Afun}, \text{env}) = v' \\
& \quad \text{set}_\text{zam}(\text{env}', \text{fenv}(\text{domain})) = \text{env}' \\
\mathcal{E}, \mathcal{F}^+ = (\sigma, \text{env}, \text{fenv}, \text{Return}(e)) = (\sigma, \text{some env}', \text{fenv}, \text{error}) & \quad \text{(EVAL-STT-RE2)} \\
\epsilon \cdot \text{env}, \text{fenv} & \leftarrow \sigma, \mathrm{bin}_{\text{for}} \times \mathcal{E}, \mathcal{F}^+ \cdot e = \epsilon_1, \epsilon_2, \ldots, \epsilon_n \\
& \quad \text{env}_c(\text{env}, \text{fenv}) = \text{true} \land \text{set}_\text{zam}(\text{Return}(e), \text{env}) = \text{Some env}' \\
& \quad (\text{repeat } \sigma \mathcal{U}_e(\sigma, \text{env})) = \epsilon_1, \epsilon_2, \ldots, \epsilon_n \sim [\mathcal{O}_1, \mathcal{O}_2, \ldots, \mathcal{O}_n] \\
& \quad \exists v, [\mathcal{O}_1, \mathcal{O}_2, \ldots, \mathcal{O}_n], \text{error} \ 	ext{Some } v_i \\
& \quad \text{set}_\text{zam}(\text{env}', \text{fenv}(\text{domain})) = \text{env}' \\
\mathcal{E}, \mathcal{F}^+ = (\sigma, \text{env}, \text{fenv}, \text{Return}(e)) = (\sigma, \text{some env}', \text{fenv}, \text{error}) & \quad \text{(EVAL-STT-RES1)} \\
\epsilon \cdot \text{env}, \text{fenv} & \leftarrow \sigma, \mathrm{bin}_{\text{for}} \times \mathcal{E}, \mathcal{F}^+ \cdot e = \epsilon_1, \epsilon_2, \ldots, \epsilon_n \\
& \quad \text{env}_c(\text{env}, \text{fenv}) = \text{true} \land \text{set}_\text{zam}(\text{Return}(e), \text{env}) = \text{Some env}' \\
& \quad (\text{repeat } \sigma \mathcal{U}_e(\sigma, \text{env})) = \epsilon_1, \epsilon_2, \ldots, \epsilon_n \sim [\mathcal{O}_1, \mathcal{O}_2, \ldots, \mathcal{O}_n] \\
\mathcal{E}, \mathcal{F}^+ = (\sigma, \text{env}, \text{fenv}, \text{Return}(e)) = (\sigma, \text{some env}', \text{fenv}, \text{error}) & \quad \text{(EVAL-STT-RES2)}
\end{align*}
\]

If the final results exist invalid value, the program will be thrown out. Otherwise valid results are obtained, the results are written in the respective \( \text{Afun} \) directly by \( \text{write}_\text{di} \). Then, the current domain of \( \text{env} \) is updated according to the domain of \( \text{fenv} \). Obviously, other statements of the function body occurring after the return statement are stopped by \( \text{env}_c(\text{check}) \).

The semantics of \( \text{Snil}, \text{Throw}, \) and \( \text{Fstop} \) statements are respectively defined by the rules, EVAL-STT-SNIL, EVAL-STT-THROW, and EVAL-STT-FUNSTOP below.

\[
\begin{align*}
\epsilon \cdot \text{env}, \text{fenv} & \leftarrow \sigma, \mathrm{bin}_{\text{for}} \times \mathcal{E}, \mathcal{F}^+ \cdot \text{true} \\
\mathcal{E}, \mathcal{F}^+ = (\sigma, \text{env}, \text{fenv}) & = (\sigma, \text{env}, \text{fenv}), \text{null} \\
& \quad \text{(EVAL-STT-SNIL)} \\
\epsilon \cdot \text{env}, \text{fenv} & \leftarrow \sigma, \mathrm{bin}_{\text{for}} \times \mathcal{E}, \mathcal{F}^+ \cdot \text{true} \\
& \quad \text{env}_c(\text{env}, \text{fenv}) = \text{true} \land \text{set}_\text{zam}(\text{Throw}(e), \text{env}) = \text{Some env}' \\
\mathcal{E}, \mathcal{F}^+ = (\sigma, \text{env}, \text{fenv}, \text{Throw}) & = (\text{null}, \text{env}', \text{fenv}, \text{stop}) \\
& \quad \text{(EVAL-STT-THROW)} \\
\epsilon \cdot \text{env}, \text{fenv} & \leftarrow \sigma, \mathrm{bin}_{\text{for}} \times \mathcal{E}, \mathcal{F}^+ \cdot \text{true} \\
& \quad \text{env}_c(\text{env}, \text{fenv}) = \text{true} \\
\mathcal{E}, \mathcal{F}^+ = (\sigma, \text{env}, \text{fenv}, \text{Fstop}) & = (\sigma, \text{fenv}, \text{fenv}, \text{stop}) \\
& \quad \text{(EVAL-STT-FUNSTOP)}
\end{align*}
\]

Because \( \text{Snil} \) represents a skip statement, its evaluation has no explicit effect and the gas does not be deducted. Meanwhile, \( \text{Throw} \) is a special kind of statement in Solidity that stops the current program and sets the memory state as \( \sigma_{\text{Init}} \). Finally, \( \text{Fstop} \) is a conventional statement in Lolisa. Therefore, \( \text{Fstop} \) also does not deduct the balance of gas and it is used to represent all executed statements in function bodies, which do not include return statements, and then reset the current \( \text{env} \) according to \( \text{fenv} \).

5.5 Development of the Standard Library and Evaluation of Programs

As mentioned previously, we have developed a small standard library in Lolisa incorporating the built-in data structures and functions of EVM to facilitate the execution and verification of Solidity programs rewritten in Lolisa using higher-order logic theorem-proving assistants directly. This subsection presents the details regarding the standard library. Then, based on the syntax, semantics, and standard library formalization, we define the semantics governing the evaluation (i.e., execution) of programs written in Lolisa.

**Standard Library** Herein, we assume that the built-in data structures and functions of EVM are correct. This is reasonable because, firstly, the present focus is on the verification of high-level smart contract applications rather than the correctness of EVM. Secondly, Lolisa is sufficiently powerful to implement any data structure or function employed by EVM. Thus, we need only implement the logic of these built-in EVM features using Lolisa based on Solidity documentation [21] to ensure that these features are well formed. For example, an address is a special compound type in Solidity that has members balance, send, and call. However, we can treat an address as a special struct type in Lolisa, and define it using Lolisa syntax, as shown Figure 9. We can then bind the identifier with the respective special address defined in Section 2. All other built-in data structures and functions of EVM are defined in a similar way. Typically, \( \text{requires} \) is a special standard function that requires no special address, and, according to Solidity documentation [21], is defined in Lolisa as follows:
requires \( \lambda (\forall t_\delta \cdot \text{type}, e : \text{expr}_{t_\delta} \text{Type}) \). If \( e \neq \text{Nil} \) (Throw).

Then, we pack all these data structures and functions together as a standard library in Lolisa, which is executed prior to executing user programs. In this way, all the built-in functions and data structures of EVM can be formalized in Lolisa, which allows the low-level behavior of EVM to be effectively simulated rather than building a formal EVM. Currently, this standard library is a small subset that includes only \text{msg}, \text{address}, \text{block}, \text{send}, \text{call}, and \text{requires}.

\[
\begin{align*}
\text{Address} & \equiv \text{Str}_{\text{type}} \_0\text{address} (\text{str}_{\text{mem}} \ \text{TInt} (\text{Nvar address})) \\
& \quad (\text{str}_{\text{mem}} \ \text{TInt} (\text{Nvar balance})) \\
& \quad (\text{str}_{\text{mem}} \ (\text{Tfid} \text{Some}_\text{Send}) (\text{Nvar send})) \\
& \quad (\text{str}_{\text{mem}} \ \text{TInt} (\text{Nvar gas} (\text{str}_{\text{NIL}}))) \_0\text{address} 2 \text{ occupy}
\end{align*}
\]

Figure 9. Address type declaration in Solidity, and its equivalent as a special struct type in Lolisa syntax

**Evaluation of Programs** The semantics governing the execution of a Lolisa program (denoted as \( P(\text{st}) \)) are defined by the rules EXE-F and EXE-IF below:

\[
\begin{align*}
\text{EXE-F:} \\
\text{EXE-IF:}
\end{align*}
\]

These rules represent two conditions of \( P(\text{st}) \) execution. Under the first condition governed by the rule EXE-F, \( P(\text{st}) \) terminates after a finite number of steps owing to the return of a \text{stop}, \text{exit}, or \text{error}. Under the second condition governed by the rule EXE-IF, \( P(\text{st}) \) cannot terminate via its internal logic, and would undergo an infinite number of steps. Therefore, \( P(\text{st}) \) is deliberately stopped via the gas-limitation-checking mechanism. Here, \( \text{opars} \) represents an optional arguments list. In addition, as discussed in Subsection 5.1, the initial environment \( \text{env} \) and super-environment \( \text{fenv} \) are equivalent, except for their gas values, which are initialized by the helper function \( \text{init}_{\text{env}} \text{gas} \), and the initial gas value of \( \text{env} \) is set by \( \text{set}_{\text{gas}} \text{gas} \). Finally, the initial memory state is set by \( \text{init}_{\text{mem}} \text{gas} \), taking \( P(\text{st}) \) and the standard library \text{lib} as arguments.

6. Validation and Case Study

While developing a set of formal syntax definitions and semantics for a real-world programming language is no small task, ensuring that the semantics faithfully capture the intended behaviors of programs written in such a language is even more difficult. The smallest error or omission in the rules governing the semantics can render the project incomplete or even incorrect. A similar work proposed a number of approaches to validate the formal semantics of languages like Lolisa [24]. The approaches given in that work included manual reviews, proving the properties of the semantics, verified translations, testing executable semantics, and equivalence with alternate validated semantics. Several of these approaches were prototyped, but not carried to completion. Therefore, these validation approaches should be considered as a work in progress. In the present study, we employ three of these approaches to validate the semantics of Lolisa: (1) proving the properties of the semantics, (2) defining equivalent computational semantics to prove the consistency between the natural semantics given in this paper and the computational semantics, and (3) implementing the formal verified interpreter based on computational semantics to execute and verify test cases.

For the first validation approach, we prove some meta-properties of type safety for Lolisa. Specifically, because Lolisa is a strongly typed language defined using GADTs, the \text{progress} and \text{preservation} properties of Lolisa expressions are easily proven in the following theorem.

**Theorem** (type safety)

1. If \( e : \text{expr}_{t_\delta} \text{Type} \) and \( e \mapsto e' \), then \( e' : \text{expr}_{t_\delta} \text{Type} \).
2. If \( e : \text{expr}_{t_\delta} \text{Type} \), then either \( e(v) \) or some \( e' \) exists such that \( e \mapsto e' \).

**Proof.** The property of \text{preservation} is proven by induction on transition defined by the rules given in Subsection 5.3. The property of \text{progress} is proven by induction on typing defined by the rules given in Subsection 3.3.

The \text{progress} and \text{preservation} properties for other components of Lolisa can be proven in a similar manner. For the second and third validation approaches, we have implemented a formal verified interpreter in Coq, denoted as FEither, which incorporates about 7000 lines of Coq code (not including proofs and comments). This interpreter is developed strictly following the formal syntax and semantics of Lolisa based on GERM. To be specific, it is implemented by computational functions (considered as the mechanized computational semantics), which is equivalent the natural semantics of Lolisa given in this paper, following the tutorial given in subsection 6.2.
of our paper [3] using Gallina which is the functional programming language provided by Coq, and it can parse the syntax of Lolisa to symbolically execute the formal programs written in Lolisa. On the one hand, we are ongoing to prove the consistency between the semantics of FEther and Lolisa. On the other hand, Ether is employed to prove the properties of real-world programs. This process is effective at exposing errors not only in the test suites that exemplify expected behaviors, but also in normal smart contracts. As a simple example, we consider the Solidity program given in Figure 10, and present its equivalent formal version, translated line-by-line and written in Lolisa, in the Figure 11. Here, it is clear that the program will be thrown out if the message sender in the index mapping list and the current time now are less than privilegeOpen or are greater than privilegeClose. This is easily proven manually with the inductive predicate semantics defined previously. Meanwhile, we can verify this property by symbolically executing the program with the help of FEther in Coq directly, as shown in Figure 12. The formal intermediate memory states obtained during the execution and verification of this Lolisa program using FEther are shown in Figure 13. Then, we can compare the mechanized verifying results and the manually obtained results to validate the semantics of Lolisa. Besides, the application of FEther based on Lolisa and GERM also certifies that our EVI theory is feasible. Work focused on certifying the interpreter correctness is ongoing. Additional details regarding the development of FEther and its correctness proofs are the subject of a following paper.

```solidity
1. function example () public payable {
2.     uint index = indexes[msg.sender];
3.     uint open;
4.     uint close;
5. ...
6.     if (privileges[msg.sender]) {
7.         open = privilegeOpen;
8.         close = privilegeClose;
9.     } else {
10.          open = ordinaryOpen;
11.          close = ordinaryClose;
12.     }
13. if (now < open || now > close) {
14.     throw();
15. }
16. if (subscription + rate > TOKEN_TARGET_AMOUNT) {
17.     throw();
18. }
19. ...
20. if (msg.value <= finalLimit) {
21.     safe.transfer(msg.value);
22.     deposits[index] += msg.value;
23.     subscription += msg.value;
24.     Transfer(msg.sender, msg.value);
25. } else {
26.     safe.transfer(finalLimit);
27.     deposits[index] += finalLimit;
28.     subscription += finalLimit;
29.     Transfer(msg.sender, finalLimit);
30.     msg.sender.transfer(msg.value - finalLimit);
31. }
32. }
```

Figure 10. An example Solidity program
1. Definition fun_fallback :
2. (Var (Some public) (Evar (Some index) Tuint));
3. (Assignv (Evar (Some index) Tuint)
4. (If (Econst (@Vmap Iaddress Tuint msg (sender >> \()) None))
5. (Assignv (Evar (Some open) Tuint) (Evar (Some privilegeOpen) Tuint));
6. (Var (Some public) (Evar (Some close) Tuint));
7. ...
8. (If (Econst (@Vmap Iaddress Tbool msg (sender >> \()) None))
9. (Assignv (Evar (Some open) Tuint) (Evar (Some privilegeClose) Tuint));
10. ...
11. ...
12. ...
13. ...
14. ...
15. ...
16. ...
17. ...
18. ...
19. ...
20. ...
21. ...
22. ...
23. ...
24. ...
25. ...
26. ...
27. ...
28. ...
29. ...
30. ...
31. ...
32. ...
33. ...
34. ...
35. ...

Figure 11. The formal version of Figure 10 written in Lolisa

```
Figure 12. Execution and verification of the Lolisa program in Figure 10 using the formal interpreter FEther in Coq
```
7. Extensibility and Universality

While ensuring that the developed set of formal syntax definitions and semantics faithfully capture the intended behaviors of programs written in real-world programming languages is essential, further ensuring that this set can be applied for multiple programming languages is also of great value. Therefore, implementing extensibility and universality in the Lolisa design was a goal considered from the beginning of its development. As such, we deliberately incorporated sufficient extensible space in Lolisa for extending features such as pointer formalization and the implementation of independent definition operators. Extensibility is further accommodated by the independence of syntax inductive predicates within the same level indicated by Convention 1, which is further supported in the semantics definitions. Therefore, Lolisa is easily extended to incorporate the features of other languages by adding new typing rule constructors in the formal abstract syntax of Lolisa.

First, the formal syntax of Lolisa is overly complex to accommodate its adoption by general users. While the syntax of Lolisa includes the same components as those employed in Solidity, it has more strict formal typing rules. Therefore, Lolisa syntax must include some additional components not supported in Solidity, such as type annotations and a monad-type option. Moreover, the syntax of Lolisa is formally defined in Coq formalization as inductive predicates. As a result, Lolisa code looks much more complicated than corresponding Solidity code, as shown in Figure 10, even though Lolisa and Solidity code present a line-by-line correspondence. An example of this difficulty is illustrated by the following code of a Taint expression $b$ being assigned to a Tint expression $a$ in Lolisa on the left, and the equivalent assignment in Solidity on the right.

\[
\text{Assignv (Evar (Some a) Taint) (Evar (Some b) Taint)); \equiv a = b;}
\]

Thus, Lolisa is obviously unfriendly for general users, and this increases the difficulty of writing code in Lolisa manually or the difficulty of developing a translator between Lolisa and Solidity or another language. Second, the Lolisa, playing the role of the core of universal formal intermediate programming language specification, can be expected to become very large after being extended, and this will increase the difficulty of adapting the formal syntax to a variety of languages if the syntax remains complex without explicit classification.

Fortunately, Coq and some other higher-order theorem-proving assistants provide a special macro-mechanism. Taking Coq as an example, this mechanism is denoted as the notation mechanism [5]. A notation is a symbolic abbreviation denoting some term or term pattern, which is parsed by Coq automatically. For example, an assignment in Lolisa can be wrapped as follows:

\[
\text{Assignv (Evar (Some 0) Tint) (Evar (Some t1) Taint)}
\]

Substituting $a$ and $b$ from the previous example yields NOTATION below, which demonstrates that the notation is nearly equivalent with the original Solidity syntax.

\[
\text{Assignv (Evar (Some a) Taint) (Evar (Some b) Taint)); \equiv "a := b" \equiv "a = b" (NOTATION)}
\]

Through this mechanism, we can hide the fixed formal syntax components, and thereby provide a simpler syntax to users. Moreover, this mechanism makes the equivalence between real-world languages and Lolisa far more intuitive. Therefore, we provide a preliminary scheme based on this macro-mechanism to improve the extensibility and universality of Lolisa systematically.
The architecture of the proposed preliminary scheme is illustrated in Figure 14. Here, we treat Lolisa as the core formal language, which is transparent for real-world users, and we logically classify the formal syntax and semantics of Lolisa according to a general component $G$ and $n$ special components $S_i$, as defined by rule 1 below. Correspondingly, a general-purpose programming language $L_i$ can be formalized by the Lolisa subset $G \cup S_i$ by wrapping the subset using notation as a symbolic abbreviation $N_i$ for $L_i$, which adopts syntax symbols that are nearly equivalent with the original syntax symbols of $L_i$. Through this method, each $L_i$ will have a respective notation set $N_i$ that satisfies $N_i \subseteq \text{Lolisa}$. This relation is defined as rule 2 below.

$$Lolisa \equiv G \cup \bigcup_{i=0}^{n} S_i \quad (1)$$

$$\forall i \in \mathbb{N} . L_i \leftrightarrow N_i \equiv G \cup S_i \quad (2)$$

As discussed, the notation layer can hide the details of the formal syntax, making it more user-friendlier and clarifying the equivalence between real-world languages and Lolisa. In addition, the symbolic abbreviation set $N$ facilitates the systemization and classification of the formal syntax and semantics of Lolisa. As such, the proposed scheme addresses the problems impeding the extensibility and universality of Lolisa discussed above. As notations, such as “(+)” and “(->)”, shown in 11, we are presently conducting the relevant work of encapsulate Lolisa according to the above architecture, and this work will be completed in the near feature.

8. Conclusion and Future Work

In this paper, we defined the formal syntax and semantics for a large subset of Solidity, which we denoted as Lolisa. To our knowledge, Lolisa is the first mechanized and validated formal syntax and semantics for Solidity. The formal syntax of Lolisa is strongly typed according to GADTs. The syntax of Lolisa includes nearly all the syntax in Solidity, and the two languages are therefore equivalent with each other. As such, Solidity programs can be translated to Lolisa line-by-line without rebuilding or abstracting, which are operations that are too complex to be conducted by general programmers, and may introduce inconsistencies. By basing the formal semantics of Lolisa on our GERM framework in conjunction with EVI, programs written in Lolisa can be, in theory, symbolically and automatically executed in higher-order theorem-proving assistants, and thereby verify the corresponding Solidity programs simultaneously. Moreover, we have mechanized Lolisa in Coq completely, and have developed a formal interpreter in Coq based on Lolisa. The formal interpreter was employed to validate the semantics of Lolisa, and certify that Lolisa satisfies the propositions of EVI. We also presented an example to demonstrate the execution and verification process of Lolisa in Coq. In addition, we validated the semantics of Lolisa using two other distinct approaches, including proving the properties of the semantics and through equivalence with alternate verified semantics. Finally, we illustrated the extensibility and universality of Lolisa, and proposed an initial scheme for systematically simplifying and extending Lolisa to support the formalization of multiple general-purpose programming languages. As a result of the present work, we can now directly verify smart contracts written in Solidity using Lolisa. In the future, we hope that Lolisa might be sufficiently powerful and friendly to be used by general programmers to verify their programs easily. Presently, we are working toward verifying the correctness of FEther, and developing a proof of the equivalence between computable semantics and inductive semantics. Subsequently, we will implement our proposed preliminary scheme based on the notation mechanism of Coq to extend Lolisa along two important avenues. First, we will seek to support other features of Solidity, such as inline assembly. Second, we will seek to support other
high-level programming languages, including Serpent [31]. Finally, we will build a general formal verification toolchain for blockchain smart contracts based on EVI to achieve the ultimate goal of automatic smart contract verification.

Acknowledgements

The authors wish to thank Marisa, Xia Yang, Yan Fang, and Yunzhuang Cai for their kind assistance.
Reference

[1] Nakamoto, Satoshi. "Bitcoin: A peer-to-peer electronic cash system." (2008): 28.
[2] Xi, Hongwei, Chiyan Chen, and Gang Chen. "Guaranteed recursive datatype constructors." ACM SIGPLAN Notices. Vol. 38. No. 1. ACM, 2003.
[3] Zheng Yang, Hang Lei, Xia Yang. “A general formal memory framework in Coq for verifying the properties of programs based on higher-order logic theorem proving with increased” https://arxiv.org/abs/1803.00403.
[4] Winskel, Glynn. The formal semantics of programming languages: an introduction. MIT press, 1993.
[5] The Coq development team. The Coq proof assistant. http://coq.inria.fr, 1999 – 2018.
[6] Arvind Narayanan, Joseph Bonneau, Edward Felten, Andrew Miller, and Steven Goldfeder. Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction. Princeton University Press, 2016.
[7] Ethereum. Ethereum solidity documentation, 2017. https://solidity.readthedocs.io/en/develop/.
[8] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. 2014. (Updated for EIP-150 in 2017) http://yellowpaper.io/.
[9] Michael del Castillo. The DAO Attacked: Code Issue Leads to $60 Million Ether Theft, 2016. https://www.coindesk.com/dao-attacked-code-issue-leads-60-million-ether-theft/, accessed on Dec 2, 2017.
[10] JD Alois. Ethereum Parity Hack May Impact ETH 500,000 or $146 Million, 2017. https://www.crowdfundinsider.com/2017/11/124200-etherum-parity-hack-may-impact-eth-500000-146-million/, accessed on Dec 2, 2017.
[11] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. Cryptology ePrint Archive, Report 2016/633, 2016. http://eprint.iacr.org/2016/633.
[12] Manticore. https://github.com/trailofbits/manticore, accessed on Dec 2, 2017.
[13] Mythril. https://github.com/b-mueller/mythril/, accessed on Dec 2, 2017.
[14] Shawn Wilkinson, Torn Boshevski, Josh Brandoff, and Vitalik Buterin. Storj a peer-to-peer cloud storage network. 2014. https://storj.io/storj.pdf.
[15] Christian Reitwiessner. Dev update: Formal methods. 2016. https://blog.ethereum.org/2016/09/01/formal-methods-roadmap/.
[16] Peter Rizzo. In formal verification push, ethereum seeks smart contract certainty. 2016. http://www.coindesk.com/ethereum-formal-verification-smart-contracts/.
[17] Wood, Gavin. "Ethereum: A secure decentralised generalised transaction ledger." Ethereum Project Yellow Paper 151 (2014): 1-32.
[18] Hildenbrandt, Everett et al. “KEVM: A Complete Semantics of the Ethereum Virtual Machine.” (2017).
[19] ©IFCA, Yoichi Hirai c. “Defining the Ethereum Virtual Machine for Interactive Theorem Provers.” (2017).
[20] Christian Reitwiessner. Dev update: Formal methods. 2016. https://blog.ethereum.org/2016/09/01/formal-methods-roadmap/.
[21] The Ethereum development team. https://solidity.readthedocs.io/en/develop/.
[22] Glynn Bird, Cloud Data Services, “Block chain technology, smart contracts and Ethereum” https://developer.ibm.com/clouddataservices/2016/05/19/block-chain-technology-smartcontracts-and-ethereum/, accessed: 2017-04-20
[23] Condit, J., Harren, M., McPeak, S., Necula, G.C., Weimer, W.: CCured in the real world. In: PLDI ’03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation, pp. 232–244. ACM Press (2003)
[24] Blazy, Sandrine, and Xavier Leroy. "Mechanized semantics for the C*light subset of the C language." Journal of Automated Reasoning 34.3 (2009): 263-288.
[25] Amani, Sidney, et al. "Towards Verifying Ethereum Smart Contract Bytecode in Isabelle/HOL.." CPP. ACM. To appear (2018).
[26] Norrish, Michael. “C formalised in HOL..” (1998).
[27] Dreyer, Derek. “Understanding and Evolving the ML Module System.” (2005).
[28] Cardelli, Luca. A semantics of multiple inheritance. In G. Kahn, D. MacQueen, and G. Plotkin, editors, Semantics of Data Types, volume 173 of Lecture Notes in Computer Science, pages 51–67. Springer-Verlag, 1984. Full version in Information and Computation, 76(2/3):138–164, 1988.
[29] Duff, T.: On Duff’s device (1988). URL http://www.lysator.liu.se/c/duffs-device.html. Message to the comp.lang.c UseNet group
[30] Wadler, Philip. “Monads for Functional Programming.” Advanced Functional Programming (1992).
[31] Ethereum. Ethereum serpent documentation, 2017. https://github.com/ethereum/wiki/wiki/Serpent.
[32] Norrish, M.: C formalised in HOL. Ph.D. thesis, University of Cambridge (1998). Technical report UCAM-CL-TR-453
[33] Leroy, Xavier. "The CompCert verified compiler." Documentation and user’s manual. INRIA Paris-Rocquencourt (2012).
[34] Appel, Andrew W. “Verified software toolchain.” ESOP. Vol. 11. 2011.
[35] Gu, Ronghui, et al. "Deep specifications and certified abstraction layers." ACM SIGPLAN Notices. Vol. 50. No. 1. ACM, 2015.
[36] Tews, H., Weber, T., Völp, M.: A formal model of memory peculiarities for the verification of low-level operating-system code. In: Proceedings of the International Workshop on Systems Software Verification (SSV’08), Electronic Notes in Computer Science, vol. 217, pp. 79–96 (2008)
[37] Igarashi, Atsushi et al. “Featherweight Java: A Minimal Core Calculus for Java and GJ.” (1999).
[38] Syme, Don. “Proving Java Type Soundness.” Formal Syntax and Semantics of Java (1999).
[39] Maffeis, Sergio et al. “An Operational Semantics for JavaScript.” APLAS (2008).
[40] Account Types, Gas, and Transactions, http://ethdocs.org/en/latest/contracts-and-transactions/account-types-gas-and-transactions.html#account-types-gas-and-transactions